UNIFORM VERIFICATION OF SAFETY FOR PARAMETERIZED NETWORKS OF HYBRID AUTOMATA

BY

TAYLOR T. JOHNSON

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:

Assistant Professor Sayan Mitra, Chair
Professor Tarek F. Abdelzaher
Professor William H. Sanders
Associate Professor Mahesh Viswanathan

Abstract

Distributed cyber-physical systems (CPS) incorporate communicating agents with their own cyber and physical states and transitions. Such systems are typically designed to accomplish tasks in the physical world. For example, the objective of a robotic swarm may be to cover an area while avoiding collisions. The combination of physical dynamics, software dynamics, and communications leads these systems naturally to be modeled as networks of hybrid automata. Hybrid automata are finite-state machines with additional real-valued continuous variables whose dynamics may vary in different states. These networks are naturally parameterized on the number of participants—processes, robots, vehicles, etc. The uniform verification problem is to verify (prove) some property regardless of the number of participants. In this dissertation, we develop a framework for formally modeling and automatically verifying networks composed of arbitrarily many participants. Three methods are presented and evaluated for proving safety properties—those that always hold throughout the evolutions of the system.

The first method is a backward search from the set of unsafe states, which are those that violate a desired safety property. The method computes the set of reachable states for a parameterized network, and checks that the intersection of the reachable states and unsafe states is empty. We apply this technique using the Model Checker Modulo Theories (MCMT) verification tool to automatically verify safe separation of aircraft in a conceptual air traffic landing protocol of the Small Aircraft Transportation System (SATS), regardless of the number of aircraft involved in the protocol.

The *Passel* verification tool we develop as a part of this dissertation implements the next two methods and can verify safety properties of parameterized networks of hybrid automata with dynamics specified as rectangular differential inclusions. The second method computes

the set of reachable states for networks composed of a finite number of participants. While this method cannot in general prove safety regardless of the number of participants, it can be used as a subroutine for other methods, such as synthesizing candidate inductive invariants to perform uniform verification. It is also useful on its own as an initial sanity check prior to attempting to prove properties regardless of the number of participants, which is harder in general—in terms of decidability and complexity.

The third method—when it is successful—automates the traditionally deductive verification approach of proving inductive invariants. This is accomplished by combining model checking with theorem proving. An algorithm synthesizes candidate inductive invariants by computing the reachable states of finite instantiations. The conditions for inductive invariance are then checked for these candidates. If a small model theorem applies, finite instantiations of the parameterized network can be checked, but in general, a theorem prover is queried. In this case, the inductive invariance conditions are encoded as satisfiability checks, for which it may be possible to discharge automatically. When these steps are successful, a fully automatic verification of safety is achieved.

The main contributions of this thesis include (a) the modeling framework for parameterized networks of hybrid automata, (b) the first fully automatic uniform verification of parameterized networks composed of hybrid automata with rectangular dynamics, such as Fischer's mutual exclusion protocol and SATS, (c) formalization of the SATS case study as a uniform verification problem, (d) the *Passel* verification tool, (e) an extension of the invisible invariants and split invariants approaches for invariant synthesis to networks of hybrid automata, and (f) a small model theorem for checking inductive invariance conditions of networks of rectangular hybrid automata.

Acknowledgments

This dissertation would not have been possible without the help, support, and encouragement of many others. I am indebted to my committee whose encouragement made possible, and criticisms sharpened, all aspects of this work. My adviser Prof. Sayan Mitra has been the most influential in every phase of this dissertation, from discussing preliminary ideas on the blackboard, to late nights of careful reading and final editorial comments on sentence structure and prose. I am extremely fortunate to have had Prof. Mitra as my adviser throughout graduate school. Prof. Tarek Abdelzaher's suggestions and feedback helped shape the cyber-physical systems aspects. Prof. William Sanders instilled in me an appreciation for the art of systems modeling. Prof. Mahesh Viswanathan's detailed understanding of the dissertation identified several problems in earlier drafts which has greatly improved the dissertation.

My family has always supported all my endeavors and shaped me into who I am today, and I am especially grateful to Mom, Dad, Brock, and Brenda. My parents sacrificed much for my education—from moving to rural Texas and to west Texas—and I am deeply grateful for them. Without my cousin Tommy's support, I probably would not have stayed in graduate school, so I am very thankful for his help. I'm thankful to have Ellen's family as a part of my extended family, and thank Bob, Gayden, Kate, Katherine, Liz, Lucy, and Mike. I have been fortunate to have been surrounded by excellent fellow group mates during my time—Sridhar, Zhenqi, Jeremy, Adam, Koushik, Karthik, Berenice, Hongxu, and Rob—from whom I've learned much. I've also been fortunate to have spent five years in the Coordinated Science Laboratory (CSL), where I've made many friends and learned much from Adel, Ahmed, Alan, Danny, Debjit, June, Leonardo, Matt, Navid, Nanjun, Neal, Puskar, Ronald, Sairaj, Shamina, Sobir, Yangmin, and so many others. I thank Profs. Marco Caccamo,

Yih-Chun Hu, Rakesh Kumar, Steve LaValle, Michael Loui, Steve Lumetta, Madhusudan Parthasarathy, Lui Sha, and Nitin Vaidya for teaching me and giving me advice. I am grateful to Prof. Daniel Liberzon who taught me almost all the control theory I know through several courses, and whose questions in the preliminary exam sharpened the remainder of the research undertaken in this dissertation. Dr. Richard Scott Erwin of the Air Force Research Lab (AFRL) has served as a valuable mentor and provided a different perspective on verification work. The staff of CSL has helped me with innumerable requests, and I am extremely thankful to have had the help of Carol, Jana, Linda, Lila, Ronda, and Angie.

I've enjoyed my five years in Champaign-Urbana due in part to friends like Alan, Seth and Amanda, Jackson and Frances, Paul and Sarah, Rakesh, Stan and Xian, Freddy, and Adrienne. I am grateful to the ECE Publications Office, particularly Jan Progen for carefully proofreading this dissertation, which has improved its consistency. I also thank the anonymous reviewers who have critiqued the work in this dissertation that has already been published. I am grateful to Cesar Muñoz for providing a PVS specification of SATS and to Karthik Manamcheri for developing an UPAAAL model of it. I am grateful to the National Science Foundation, the Air Force Office of Scientific Research, and The Boeing Company for providing the financial support for this research. This dissertation is based upon our papers [1–3], which were supported by the National Science Foundation under CAREER Grant No. 1054247.

I am eternally grateful to my beautiful wife Ellen, who has quite literally taken care of me during the past couple years, and especially the last year. Without Ellen I would not have finished, nor would I be as happy as I am today, so with love, *thank you*. Finally, I would like to thank everyone else with whom I interact, as I'm sure I've failed to mention everyone explicitly.

Contents

Chapter 1

Introduction

Cyber-physical systems (CPS) involve coordination of software and physics through the use of control, computation, and communication. Distributed cyber-physical systems (DCPS) have geographically distributed components, and therefore, issues like message delays, asynchrony, and failures make their design and analysis challenging. Examples of DCPS arise in air traffic control where aircraft communicate with one another and traffic controllers to safely take-off and land, robotics where robots coordinate to solve some task in the physical world like observing a region while avoiding collisions, and many other domains. To aid in the design and development of such systems, we present algorithmic verification techniques for DCPS in this dissertation.

1.1 Motivation and Background

In many engineering domains, cyber-physical systems (CPS) are becoming common as software enables higher levels of autonomy. For instance, unmanned aerial vehicles are beginning to share airspace with commercial and passenger air traffic [4], autonomous satellites will soon service aging satellites [5], networked medical devices are being worn by and implanted in humans [6], and cars may soon drive themselves [7]. As the autonomy of such systems increases, we need methods for helping ensure their design correctness and safe operation, as any of these safety-critical systems has the potential to cause catastrophic failure. Many current and future CPS—such as in automotive and air traffic control protocols—involve a complex interaction between software state of many independent agents to ensure physical safety.

Hybrid systems modeling frameworks (e.g., hybrid automata [8, 9], and see more recent

proceedings of HSCC [10–12]) combine continuous and discrete evolution and have become a standard formalism for modeling software systems interacting with the physical world. In systems where many nearly identical automata interact, hybrid automata models parameterized by automaton identifiers preserve the symmetry arising from the repeated structure. Systems exhibiting this pattern abound around us—from MAC protocols, air-traffic control systems, and real-time distributed algorithms, to control systems for robotic swarms and cell arrays in tissue. Additionally, each robot in a robotic swarm is naturally modeled as a hybrid automaton, receiving messages from nearby neighbors to determine new control polices to accomplish some global problem like covering a physical region. Each mote in a wireless sensor network (WSN) can be modeled as a hybrid automaton, albeit the global task may be to aggregate cyber states of physical measurements, instead of coordinating physical state. Similarly, subcomponents of these systems like clock synchronization algorithms, mutual exclusion algorithms, leader election protocols, analog-to-digital converters, digital-to-analog converters, actuators, sensors, etc., can also be modeled as networks of hybrid automata, and have their own correctness specifications that may be verified. We call such systems distributed cyber-physical systems (DCPS) due to this distributed interaction of cyber and physical state.

Uniform Verification for Parameterized Networks. In such parameterized models, each automaton is an instance of a hybrid automaton template $\mathcal{A}(\mathcal{N}, i)$ that interacts with others only through shared discrete transitions (and not through continuous signals). Although the specifications of all the automata in a system are identical, modulo their identifiers, in a given execution of the system, the automata may behave differently. The composed network has both discrete and continuous dynamics, and the communication topology between automata may itself evolve over time. When the number of participants in the network is not fixed a priori, it is referred to as a *parameterized network* of instances of the template $\mathcal{A}(\mathcal{N}, i)$, and we denote the network $\mathcal{A}^{\mathcal{N}}$. Many approaches can be used for analyzing such networks, such as simulation to see if they violate some specification. The specification describes what the network should or should not do, and we focus only on safety properties—those that should always hold—in this dissertation. However, proving

that parameterized networks satisfy a safety property regardless of the number of participants cannot in general be solved using simulation. In fact, proving that even a single hybrid automaton satisfies a safety property cannot in general be solved using simulation for a variety of reasons, such as uncountability of the reals. Thus, other techniques like manual analysis, computer-aided analysis using interactive theorem provers, or model checking may be employed. These techniques also have limitations as discussed shortly.

We address the following problem in this dissertation. Given an automaton template $\mathcal{A}(\mathcal{N}, i)$ and a property $\zeta(\mathcal{N})$, the *uniform verification problem* is to prove $\forall \mathcal{N} \in \mathbb{N}$ that the parameterized network $\mathcal{A}^{\mathcal{N}} \triangleq \mathcal{A}_1 \| \mathcal{A}_2 \| \ldots \| \mathcal{A}_{\mathcal{N}}$ satisfies $\zeta(\mathcal{N})$ [13, Ch. 15]. The uniform verification problem is an infinite state verification problem. Such a formulation allows one to conclude—irrespective of the number of components $\mathcal{N}$ involved—whether a network composed of $\mathcal{N}$ instances of $\mathcal{A}(i)$ satisfies the property $\zeta(\mathcal{N})$. For instance, in a realization of the automated highway system, how could one automatically verify that any interaction between an arbitrary number of cars does not result in a collision? A variety of other interacting DCPS are naturally modeled as parameterized systems, such as automotive traffic protocols [14], swarm robotics and coordination [15, 16], industrial systems [17], and networked medical devices.

State-Space Explosion. One of the main challenges in applying automatic verification techniques to (even purely discrete) systems is the *state-space explosion problem* [13]. For concurrent systems like distributed algorithms, this problem is exacerbated since the growth of the state-space is exponential in the number of components (processes). In general, the composition of N processes each modeled with k states yields a product state-space with k^{N} elements.[1]

For example, consider Dijkstra's classic mutual exclusion algorithm [18] in Figure 1.1. This algorithm has a global variable k taking values in $[\mathsf{N}]_\perp$ (initially arbitrary), a local variable $l[i]$ of type $[\mathsf{N}]_\perp$ (initially arbitrary), two local Boolean variables $b[i]$ and $c[i]$ (both initially true), and five program locations for each process. Even with a shared-memory

[1] Throughout this dissertation, we use $[\mathsf{N}]_\perp \triangleq \{1, \ldots, \mathsf{N}\} \cup \{\perp\}$ for a process (or automaton) index set, where $\perp$ is a symbol not equal to any identifier. When the number of processes is not fixed—that is, when considering arbitrarily many processes—the symbol $\mathcal{N}$ and set $[\mathcal{N}]_\perp$ are used.

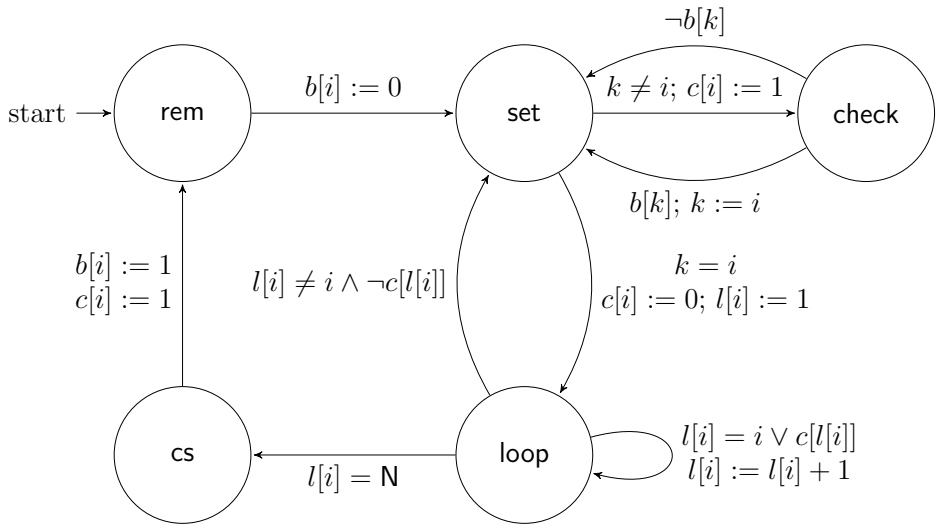

Figure 1.1: Dijkstra's mutual exclusion algorithm for process $i \in [\mathsf{N}]$ for illustrating the state-space explosion problem for concurrent discrete-state systems.

model where event-ordering interleavings like message sends and receives are not considered, this program's purely discrete state-space has $2^{2\mathsf{N}}(2(\mathsf{N}+1))^{\mathsf{N}}(5\mathsf{N})^{\mathsf{N}}$ elements, and for $\mathsf{N} = 4$, the state-space has over ten billion elements ($> 2^{33}$). A human designer would consider some small number of these states, in part based on experience and ingenuity, and also by only considering the subset of the state-space that may be reached. Much progress in computer-aided verification over the last thirty-plus years has been to incorporate such human ingenuity into clever abstractions, so algorithms need only explore small equivalence classes in these huge spaces.

Verification of DCPS must cope with the complexities of large discrete state-spaces, along with the additional modeling and computational complexity of physical state. Physical states like positions and velocities are often naturally modeled as real variables that evolve according to some ordinary differential equations (ODEs) or inclusions. Such modeling yields hybrid systems representations of DCPS with combinations of continuous and discrete states and transitions [9, 19–23]. Even relatively simple timed distributed mutual exclusion algorithms like Fischer's mutual exclusion algorithm [24] in Figure 1.2 have large discrete state-spaces. Fischer uses a global variable g taking values in $[\mathsf{N}]_\perp$ (initially $\perp$) and has four control locations for each process, yielding $(\mathsf{N}+1)(4\mathsf{N})^{\mathsf{N}}$ discrete states, along with one continuous variable $x[i]$ for each of the N processes. For $\mathsf{N} = 6$, this is already over a billion discrete

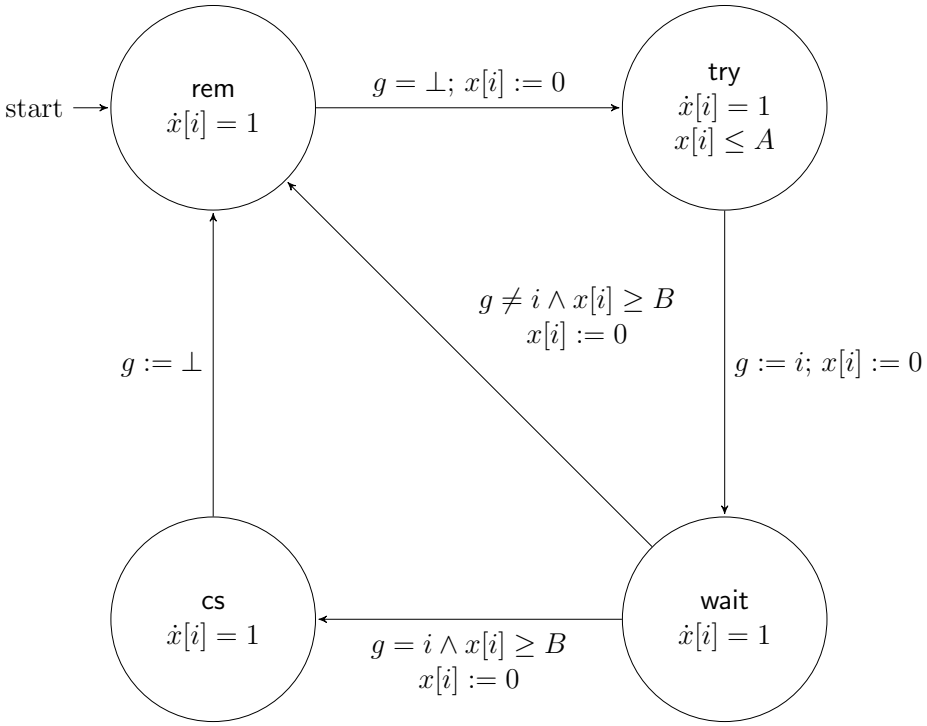

Figure 1.2: Fischer's mutual exclusion algorithm (Fischer-Timed) for process $i \in [\mathcal{N}]$.

states ($> 2^{30}$) with a continuous state-space of $\mathbb{R}^6$.

DCPS are naturally modeled as interacting networks of hybrid automata. In this dissertation, we develop a formal modeling framework for networks of hybrid automata in Chapter 2. For example, a simplified model of each aircraft in the Small Aircraft Transportation System (SATS) [2, 25–27]—an aircraft landing protocol—is shown in Figure 1.3. In this protocol, there is a global variable g taking values in $[\mathsf{N}]_\bot$ that tracks the last aircraft to enter the landing protocol, a variable $n[i]$ taking values in $[\mathsf{N}]_\bot$ used to track the aircraft immediately ahead of aircraft i (if any), and a continuous real variable $x[i]$ measuring the distance of the i^{th} aircraft from the start of the base location. The parameters L_B and L_G are real constants representing, respectively, the length of the base location and the minimum distance any aircraft in the base location must have traveled prior to any subsequent aircraft being allowed to enter the base location, where L_G is chosen large enough to ensure no two aircraft in the base location collide.

Undecidability. When uniform verification of safety properties for parameterized networks is approached with model checking, it is an infinite-state model checking problem and

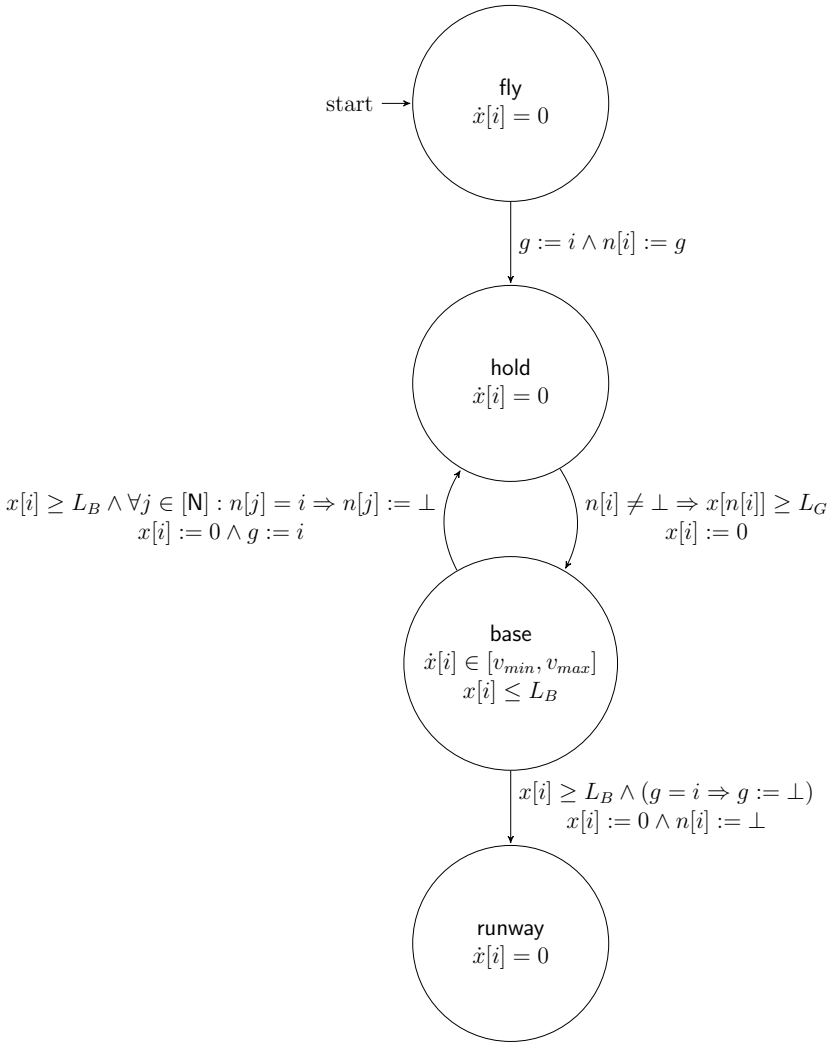

Figure 1.3: Simplified Small Aircraft Transportation System (SATS) aircraft landing protocol (SSATS) for aircraft $i \in [\mathcal{N}]$.

is undecidable, even when $\mathcal{A}(\mathcal{N}, i)$ is a finite state automaton [28, 29]. However, it has some decidable subclasses as we review in more detail in Section 1.3. To remain decidable, restrictions must be placed on $\mathcal{A}(\mathcal{N}, i)$, the parallel composition operator $\|$ (and hence how the $\mathcal{A}(\mathcal{N}, i)$'s may communicate), and the property $\zeta(\mathcal{N})$. For general classes of hybrid systems, checking if even a single $\mathcal{A}(\mathcal{N}, i)$ satisfies $\zeta(\mathcal{N})$ is undecidable, but it also has several decidable subclasses [30], such as when restrictions are placed on the continuous dynamics—like in initialized rectangular hybrid automata (IRHA)—or on the discrete dynamics [31].

Advantages of the Uniform Verification Perspective. There are large classes of DCPS and that can be expressed in the hybrid automata modeling framework, but the state-space explosion problem is exacerbated for networks of such automata. This dissertation partially addresses the uniform verification problem for parameterized networks of hybrid automata. The primary contributions of this dissertation are applications and extensions of existing uniform verification approaches developed for discrete systems to DCPS that also have continuous evolution specified by ordinary differential equations and inclusions. If successful, uniform verification proves a parameterized network satisfies a specification regardless of the number of participants in the network. Since traditional automatic verification—for example, model checking—of distributed computing systems requires picking a finite instantiation N, it is necessarily incomplete and cannot establish correctness for any choice $\mathcal{N}$.[2]

Uniform verification is also a way enable scalable verification and ameliorate the state-space explosion problem. It may be infeasible to use traditional verification to verify the largest realistic instantiation. For example, in Dijkstra's mutual exclusion algorithm, the state-space has $> 2^{424}$ elements for $\mathsf{N} = 30$, while it is reasonable that up to 30 processes or threads actually need to access some shared resource. Thus, if uniform verification succeeds, these large but finite instantiations need not be explicitly considered.

1.2 Dissertation Summary and Contributions

We present next a detailed summary of the dissertation, along with a summary of the verification techniques and other contributions it makes.

1.2.1 Modeling Framework and the *Passel* Verification Tool

Chapters 2 and 7 present, respectively, a modeling framework for parameterized networks of hybrid automata, and the *Passel* verification tool for automatically verifying safety properties of such systems. The modeling framework is a key contribution of this dissertation,

[2]Without additional analysis showing some particular finite choice N_0 is sufficiently large, so that $\mathcal{N} > \mathsf{N}_0$ need not be considered. This is one technique we exploit in this dissertation.

as it is the first such modeling framework for parameterized networks of hybrid automata with a focus on being amenable to analysis, simulation, and transformation with automated tools. *Passel* relies on the methods for proving safety properties of such networks as described in Chapters 4, 5, and 6. *Passel* represents a large undertaking of the dissertation, and is one of the main contributions.[3] To the best of our knowledge, *Passel* is the only tool for automatically verifying safety properties of parameterized networks of hybrid automata. However, MCMT [32,33] can be used for verifying parameterized networks of timed automata, and theorem prover-based techniques may allow some or full automation for some examples [34,35]. No other tools for verifying timed or hybrid automata—e.g., Charon [36], HyTech [37], UPAAAL [38], PHAVer [39], SpaceEx [40], etc.—can solve the uniform verification problem addressed in this dissertation. The methods *Passel* uses are summarized in the next three subsections.

1.2.2 Reachability Using Anonymized States for Finite Instantiations of Hybrid Automata Networks

Chapter 4 presents a method for computing the set of reachable states of finite instantiations of N automata for parameterized networks $\mathcal{A}^{\mathsf{N}}$. The method avoids computing all the permutations of automata indices, and is able to speed and scale reachability computations greatly in practice. The method is shown to be sound and to satisfy several invariants during the computation of the efficiently represented reachable states. Experimental results using the method implemented in *Passel* have enabled the computation of the reachable states for networks with hundreds of processes, whereas existing hybrid systems model checkers like PHAVer ran out of memory with at most tens of processes. Thus, the representation allows for analyzing networks orders of magnitudes larger than existing approaches.

[3] The *Passel* verification tool, along with all the examples described in this dissertation and others, is available for download: https://publish.illinois.edu/passel-tool/. If this link ever becomes outdated, please visit: http://www.taylortjohnson.com.

1.2.3 Proving Inductive Invariants for Parameterized Networks of Hybrid Automata

Chapter 5 presents a small model theorem for reducing the uniform verification problem to checking finite instantiations, assuming an inductive invariant sufficient to establish a desired safety property is provided. The theorem establishes a bound N_0—whose value is a function of the protocol and the property to be verified—such that, if any instance of the network $\mathcal{A}^{\mathcal{N}}$ of size $\mathcal{N}$ violates a safety property $\zeta(\mathcal{N})$, then an instantiation of size $\mathsf{N} \leq \mathsf{N}_0$ must also violate $\zeta(\mathsf{N})$. Thus, if no instance of size $1 \leq \mathsf{N} \leq \mathsf{N}_0$ violates $\zeta(\mathsf{N})$, one can conclude $\zeta(\mathcal{N})$ holds for all $\mathcal{N} \in \mathbb{N}$.

Such theorems were originally developed for enabling automatic deductive verification of inductive invariants for purely discrete systems [41,42]. To enable verification of some property $\zeta(\mathcal{N})$, a strong enough invariant $\Gamma(\mathcal{N})$ must be supplied that is inductive and implies $\zeta(\mathcal{N})$. Usually, the process of strengthening $\Gamma(\mathcal{N})$ until it is inductive is a manual process, but we next describe the methods we investigate in this dissertation for automatically finding proofs of inductive invariance for parameterized networks.

1.2.4 Finding Inductive Invariants for Parameterized Networks of Hybrid Automata

The weakness of the method of inductive invariance—even when a small model theorem for the class of systems and properties is available—is that it requires coming up with an inductive invariant strong enough to prove a desired safety property, which in general, requires manual intervention. Thus, a method for coming up with candidate inductive invariants can help automate the verification process.

Invisible and Split Invariants for Parameterized Networks of Hybrid Automata. In [41,42], small model theorems were combined with the *invisible invariant* method, which is a heuristic for doing such strengthening automatically. The weakness of this method is that it is a heuristic and may fail to find a suitable candidate inductive invariant strong enough to prove a desired safety property. In Chapter 6, an extension of the invisible invariant method to timed and hybrid systems is presented, along with a more general procedure that

is guaranteed to generate the best candidate invariant of a particular class, described in more detail in Section 6.4.

Abstractly, the procedure projects the reachable states onto a smaller space—the states of $\mathcal{A}^{\mathsf{P}}$, a networked composed of P automata—then lifts them up to a larger space—the states of $\mathcal{A}^{\mathcal{N}}$, a network composed of an arbitrary number of automata. However, the procedure is just a heuristic and hence incomplete [43], and fails to generate inductive invariants for many protocols in practice [41,42]. It surprisingly succeeds in generating quantified inductive invariants for some examples. A complete method—called the *split invariant* method—for synthesizing the strongest inductive invariant is presented for discrete systems in [43]. The method is based on the non-interference properties in compositional analysis [44].

We extend the invisible invariant method [41, 42, 45, 46] to hybrid automata networks in Chapter 6 and implement it in *Passel*. We use the method to automatically verify several timed and hybrid examples like Fischer's mutual exclusion protocol, a part of the Small Aircraft Transportation System (SATS) landing protocol [2,25,26], along with several other timed examples as detailed in Section 7.8. We also successfully verify several safety properties completely automatically in the purely discrete examples the method was originally applied to in [41, 42] as a sanity check. These experiments are—to the best of our knowledge—the first fully automatic verification of parameterized networks of hybrid automata, beyond the special subclass of timed automata [32, 33, 47–50].

Due to the theoretical and practical incompleteness of the invisible invariant method, we explore in Chapter 6 the theoretical and practical benefits of using the split invariant method [43] for hybrid automata networks. This method utilizes a fixed-point procedure to compute candidate invariants. Like the invisible invariant method, this generates a formula $\Gamma(\mathcal{N}) \triangleq \forall i_1, \ldots, i_{\mathsf{P}} \in [\mathcal{N}] : i_1 \neq \ldots \neq i_{\mathsf{P}} \Rightarrow \phi(i_1, \ldots, i_{\mathsf{P}})$. It may generate an inductive invariant that proves the desired safety property, and in practice, many protocols' safety properties can be proved with inductive invariants of the form $\forall i_1, \ldots, i_{\mathsf{P}} \in [\mathcal{N}] : i_1 \neq \ldots \neq i_{\mathsf{P}} \Rightarrow \phi(i_1, \ldots, i_{\mathsf{P}})$. However, some protocols need assertions with quantifier alternation like $\exists i \in [\mathcal{N}]\ \forall j \in [\mathcal{N}]$.

1.2.5 Parameterized Reachability Analysis: A Case Study on Distributed Air Traffic Control

Chapter 3 describes a backward reachability semi-algorithm for verifying parameterized networks. This general methodology is widely used in a variety of research work and software tools, such as UNDIP [51, 52], MCMT [32, 33, 53, 54], SAFARI [55], and Cubicle [56]. We present a detailed summary of the method, along with its use in several examples, including a slightly simplified version of the Small Aircraft Transportation System (SATS) landing protocol [2, 25, 26]. We use the MCMT tool for analyzing SATS. The SATS protocol is simplified in part due to limitations of available tools. For example, MCMT is the only available tool that supports any form of continuous dynamics for parameterized networks, and it is limited to timed dynamics [32, 33], which are a special subclass of the rectangular differential inclusion dynamics in the SATS protocol [25–27]. Such limitations are not only in available tools, but also in the theoretical basis, which we have addressed partially in this dissertation in Chapter 2 by developing a modeling framework allowing for more general dynamics. We have addressed tool availability by developing and releasing the software tool *Passel* that supports automatic verification of parameterized hybrid automata with continuous dynamics specified as rectangular differential inclusions. The limitations of this method and the restriction to timed dynamics serve as the motivations for the other methods we develop later in this dissertation, such as the reachability method of Chapter 4, the inductive invariance method of Chapter 5, and the invariant synthesis methods of Chapter 6.

1.3 Literature Review

Verification of cyber-physical systems (CPS) and hybrid models have enjoyed the attention of several researchers over two decades (see, for example, the recent proceedings of HSCC [10–12] for recent developments).

1.3.1 Modeling and Verification of Distributed Cyber-Physical Systems

Several formalisms have been developed for modeling and verifying CPS, such as hybrid automata [8, 19, 36], hybrid programs [14, 35, 57, 58], and hybrid input/output automata (HIOA) [9, 23]. Generally speaking, automation of analysis—such as decidability of model checking—is usually gained at the expense of less expressiveness. Thus, techniques either focus on a restricted formalism such as initialized rectangular hybrid automata (IRHA) [30] or require some level of human intervention.

Deductive verification of safety critical traffic protocols like those seen in automotive and aerospace systems has been researched in [15, 59–61]. For instance, techniques based on optimal control are used for verification of conflict resolution maneuvers in [62, 63] and automatic landing systems in [64]. Using logical techniques, curved flight maneuvers have also been verified in [58]. Automotive protocols like those that may play a role in the automated highway system have been modeled and verified semi-automatically using theorem provers [14].

Previous work on formally modeling and analyzing the Small Aircraft Transportation System (SATS)—a distributed air traffic landing protocol—has relied on using verification of purely discrete models [65–68]. Discrete abstractions capturing all behaviors of SATS are created in [66, 68]. The properties verified in these works include that there are at most four aircraft on the approach to the runway, and similar properties limiting the number of aircraft in certain zones. In [27], assuming this limited number of aircraft in the system, the authors automatically generate a set of lemmas corresponding to every combination of aircraft, and then discharge these lemmas semi-automatically in the PVS theorem prover, thus verifying the separation assurance safety property of the hybrid system. A more detailed hybrid systems model of SATS is developed and verified in [69].

Parts of these works relied on deductive verification, such as through the use of the interactive theorem prover PVS [70], supplemented with some automatic state-space exploration [27]. In [27], for example, it is first shown using PVS that SATS can have at most four approaching aircraft, and then all automatic state exploration uses this fixed number of aircraft. Traditional model checking would require the number of aircraft involved in the

system to be fixed to a natural number prior to computing the composition. For example, finite instances of a simplified version of the SATS protocol have been verified using HyTech and HARE in [71]. In contrast, we present a parameterized model of a simplified version of SATS in Chapter 3, Section 3.2 and automatically verify the key safety property of the protocol *regardless of the number of aircraft in the system*. The techniques we use could help scale verification of automated highways, peer-to-peer protocols, and other general aviation systems, such as landing protocols for unmanned aerial vehicles (UAVs).

1.3.2 Uniform Verification of Discrete Parameterized Systems

Uniform verification of parameterized systems aims to prove properties regardless of the number of participating agents. An overview of automatic approaches for verification of discrete parameterized systems appears in [13, Ch. 15] and the survey [72]. Most early works focused on asynchronous models [73,74]. The uniform verification problem is in general undecidable, even for networks of finite-state automata [28, 29]. Due to these decidability results, many works focus on incomplete, but sound, approaches for verifying parameterized systems. However, for restricted classes of systems under various communications constraints, the problem has been shown to be decidable [75].

Counter and Environment Abstractions. Several papers present sound but incomplete methods for uniform verification of parameterized discrete systems. Much existing work on uniform verification relies on computing abstractions of given system models. One abstraction is called *counter abstraction* [13, Ch. 15]. For a network of finite automata, the intuition is to count the number of processes in each discrete location, and instead of keeping track of which process is in which discrete state, simply count the number of processes in that state. Then for each counter, bound the number of processes to track, where usually the domain will now be zero, one, two, or greater than two, instead of all natural numbers, and a finite abstraction has been created. An alternative to bounding the counters is to not specify a bound, as in [76], and use model checkers developed for infinite state systems like HyTech [37]. These approaches have been used for verifying cache-coherence protocols [76,77], which have received considerable attention from research on parameterized

systems [53, 78].

However, these approaches limit the ability to check many liveness properties, as the indices of processes have been abstracted, and we can no longer tell if some particular process is doing something to make progress. An extension to this is known as the counter plus one abstraction, which keeps track of the exact state of a single process in addition to using the counter abstraction for all other processes describe above [79]. This allows checking for liveness properties of that one process, such as if it is waiting to enter the critical section, it eventually does so. A generalization of counter abstraction is *environment abstraction*, which counts the number of processes satisfying some predicates instead of those in some discrete state [80].

A generalization of counter abstraction is monotonic abstraction [81]. The intuition is the same as counter abstraction, but also an ordering on any configurations is ensured and guarantees termination of a fixed-point algorithm. This abstraction is what has been employed in a counterexample-guided abstraction and refinement (CEGAR) technique for parameterized systems [82]. The abstraction is defined in terms of the ordering, so when the ordering is refined, the system changes and a counterexample is ensured to be removed, yielding termination. Other approaches to model checking even more general classes of discrete systems—of which discrete parameterized systems are a subclass—are regular model checking [83] and omega regular model checking [84].

Network Invariants. Network invariants are an abstraction approach introduced in [85] and studied extensively [49, 86–92]. The idea is to abstract $\mathcal{N} - 1$ of the processes into one process $\mathcal{I}$—called the network invariant—that is independent of $\mathcal{N}$. When possible, the problem of proving for all $\mathcal{N} \in \mathbb{N}$ that $\mathcal{A}^{\mathcal{N}} \models \zeta(\mathcal{N})$ is reduced to proving $\mathcal{A}_1 \| \mathcal{I} \models \zeta$, which may require modifying ζ according to the abstraction that defines $\mathcal{I}$ as well. Network invariants for real-time systems are developed in [92], and a formalization of process calculi methods in timed CSP used in the Isabelle/HOL theorem prover [93] is developed in [94], with more details on both available in the thesis [50]. Network invariants have also been developed for parameterized networks of timed automata in [49].

Invisible Invariants and Small Model Theorems. Finding invariants was automated with the invisible invariants approach [41, 42], which provides a heuristic method to automatically compute inductive invariants, such as implemented in IIV [45]. Parameterized verification of liveness properties has also been investigated using an extension of invisible invariants [95–98] A small model theorem similar the one we present in Chapter 5—from our paper [3]—was introduced in [41, 42] for a class of discrete parameterized systems with bounded data, but the main difference is that our result applies to hybrid and timed systems. We can view cutoffs—an instantiation of the network that has all the behaviors of additional compositions—like those in [99–102] like small model results, in that it is sufficient to check the composition of a protocol up to the cutoff or small model bound to verify the parameterized specification. An alternative approach is [102], which computes cutoffs to verify parameterized systems, where a cutoff is an instantiation of the system that has all the behaviors of additional compositions, and it turns out that in practice, many systems have finite cutoffs. The Golok tool [102] computes cutoffs to verify a class of parameterized systems, where a cutoff is an instantiation of the system that has all the behaviors of additional compositions, and it turns out that in practice, many systems have finite cutoffs.

1.3.3 Uniform Verification of Timed and Hybrid Parameterized Systems

There are several approached for uniform verification and timed and hybrid models that require different degrees of human intervention [14,27,35,69,103,104]. There are several works addressing uniform verification for networks of the special subclass of timed automata [32, 33, 47–50, 92, 105, 106]. Uniform verification of hybrid automata networks is useful to show, for instance, that for arbitrarily many aircraft participating in a given distributed air traffic control protocol like the Small Aircraft Transportation System (SATS), no two aircraft ever collide [2, 27, 69].

Many techniques for verification of parameterized discrete systems have been applied to parameterized timed networks. An abstraction for a discretized version of Fischer's mutual exclusion [107] is created in [108]. Uniform verification of parameterized timed networks has been studied in [47, 48, 109], where a control-state reachability problem was shown to

be decidable for a restricted class of timed parameterized systems. The method combines counter abstraction with the clock regions of [110] to create a simulation relation between regions of the continuous state-space and a finite state machine. If the timed automata have more than a single real-valued clock, then checking safety properties is undecidable using a reduction to two-counter machines [48]. However, if the clocks are discrete-valued, each automaton may have any finite number of clocks. The previous undecidability result prevents using the standard initialized rectangular hybrid automata (IRHA)-to-timed automata conversion algorithm [8, 30] because it adds two clocks for every continuous variable evolving with rectangular dynamics. However, decidability of parameterized verification for IRHA could be argued using a reduction that does not involve creating multiple clocks for every real-valued continuous variable, such as the discretization of time used in [111], and combining decidability of parameterized networks with multiple discrete-valued clocks [48].

If the timed automata have urgent transitions, then checking safety properties is undecidable [47]. There are further restrictions on how the processes may communicate, as well as on the allowable guards. While checking general liveness properties is undecidable for these networks [47], some recent work develops methods for checking some liveness properties [33]. Bounded reachability methods are developed for timed parameterized systems in [32, 33], and we applied these reachability techniques to a simplified model of an air traffic control protocol in [2], reprinted in Chapter 3.

An alternative approach for uniform verification uses interactive theorem proving. The system model and the properties are specified as a theory in the language of the theorem prover, and then these properties are discharged by invoking theorem prover commands on the proof goals. The granularity of these commands and the degree of automation varies from one system to another, but proving sophisticated invariant properties requires significant manual work. This approach has been successfully applied to verify: (a) protocols modeled with timed automata [34, 112] in PVS [70], (b) SATS [27, 68, 69] in PVS, (c) Fischer's mutual exclusion protocol [103] in SAL [113], (d) aircraft separation assurance in conflict avoidance maneuvers [58] in KeYmaera [35, 104], (e) automotive collision avoidance in adaptive cruise control [14] in KeYmaera, and (f) timed parameterized systems such as an operating system scheduler [50] in Isabelle/HOL [93].

Quantified differential dynamic logic can be used to model parameterized systems, although at the expensive of sometimes requiring manual intervention [35]. In addition, the quantified differential invariants have the same shape as the class of assertions we are attempting to automatically generate for hybrid automata networks [104]. Despite these techniques using automated theorem provers being partially manual, the strengths of deductive methods are that: (a) they may be able to handle nonlinear continuous dynamics and complex discrete dynamics with data structures, and (b) they may be used to specify and verify liveness properties.

1.3.4 Symmetry Methods and Efficient State-Space Representations

Analysis and state-space construction methods that exploit symmetries have been thoroughly investigated for many classes of systems, as such methods ameliorate the state-space explosion problem [114–122]. Several symmetry methods have been developed and implemented for the Murφ verification system [123] for discrete systems. The *scalarset* data structure, which is a finite unordered set, is developed and added to Murφ in [115], and was one of the first approaches of automatically detecting and exploiting symmetries in model checking. The *repetitive id* data structure is applied to several discrete parameterized systems like cache coherence protocols in [124].

Advances in tools like UPAAAL [38] that exploit symmetries of the state-space to reduce its size vastly have enabled scaling to larger instantiations. For instance, the scalar set technique developed for Murφ was extended for timed systems and implemented in UPAAAL [125,126]. Data structures like binary decision diagrams (BDDs) [127] and advances in satisfiability (SAT) algorithms [128] have helped enable automatic verification of discrete systems with large state-spaces. Analogously, data structures like difference bound matrices (DBMs) [129–131], difference decision diagrams (DDDs) [132], and zonotopes [133,134] have helped enable automatic verification of systems with large continuous state-spaces.

1.4 Dissertation Outline

We present a modeling framework for hybrid networks and a collection of verification techniques for this framework that rely on computing reach sets, checking inductive invariants, and exploiting small model properties of the specifications. The techniques are supported in a software tool called *Passel* that we have developed. We have also performed detailed experimental analysis of our techniques on several case studies and compared it with other verification approaches and tools.

Chapter 2 develops a general formal modeling framework for parameterized networks of hybrid automata. We also formalize the uniform verification problem for such networks. It describes the syntax of a restricted class of first-order logic formulas used for specifying components of a template automaton $\mathcal{A}(\mathcal{N}, i)$, and specifies the semantics of finite compositions $\mathcal{A}^{\mathsf{N}}$ of N automata and compositions $\mathcal{A}^{\mathcal{N}}$ of arbitrarily many automata. Several examples are described that fit into this modeling framework, with their syntactic specifications described using this restricted class of formulas in the *Passel* verification tool input syntax. Chapter 2 is based in part on our previous work [1–3].

In Chapter 3, we utilize reachability techniques developed for array-based systems [53] to parameterized networks of timed automata. We present a case study of the Small Aircraft Transportation System (SATS) as a parameterized network, and automatically verify several safety properties for it that were previously semi-manually verified in other works [27, 69]. For this, we use an existing reachability tool called the Model Checker Modulo Theories (MCMT) [32, 135]. Chapter 3 is based in part on our previous work [2].

Chapter 4 presents a method for computing the set of reachable states of finite *instantiations* of parameterized networks of hybrid automata. That is, the method computes the reachable states of a fixed instance of a parameterized network, for example, $N = 17$. The method exploits symmetry in the automaton specification and reachable states to avoid considering permutations of states that are equivalent modulo automaton indices. It has been useful in scaling up reachability computations of fixed instances, e.g., for use in the invariant synthesis procedure of Chapter 6 or for performing verification using the small model theorem of Chapter 5.

In Chapter 5, we present an extension of small model theorems developed for discrete systems [41, 42] to parameterized networks of timed and hybrid automata ($\dot{x}[i] \in [a, b]$ for real constants $a \leq b$). Small model theorems allow for verification of networks composed of arbitrarily many participants using finite instantiations, and require the participants to be syntactically specified in a restrictive syntax, as the theorems are technically about the size of satisfying assignments (models) for syntactically restricted first-order logic formulas. We apply the result to prove safety properties of several examples, such as the Fischer mutual exclusion algorithm and SATS. Chapter 5 is based in part on our previous work [3].

Chapter 6 describes invariant synthesis procedures we develop and implement in *Passel*. The methods are based on the invisible invariants technique [41–43, 45, 46], but extended to networks of hybrid automata. The methods compute the set of reachable states for finite instantiations of the network, then transform an assertion representing the reachable states into a candidate inductive invariant for the parameterized (arbitrarily large) network.

Chapter 7 presents an overview of the *Passel* software tool for automatically verifying DCPS using the methodology we present in this dissertation, along with some design and implementation choices. *Passel* uses the satisfiability modulo theories (SMT) solver Z3 [136] for checking satisfiability (and validity) of formulas. *Passel* implements the reachability method of Chapter 4 that exploits symmetries in the automaton specification. *Passel* proves safety properties by automatically checking inductive invariance conditions, as described in Chapter 5. *Passel* implements the invariant synthesis procedures of Chapter 6, which can enable fully automatic verification. We present experimental results of *Passel* using the techniques of Chapters 4, 5, and 6 in Sections 7.6, 7.7, and 7.8, respectively.

Chapter 8 concludes the dissertation with a summary of the work and results, a brief discussion, and directions for future research and potential applications.

1.5 Copyright Acknowledgments

We include the following required copyright statements for permission to reprint portions of [1–3].

- For portions of [3] reproduced in this dissertation, we acknowledge the IFIP copyright:

-

-

Chapter 2

A Modeling Framework for Hybrid Automata Networks

In this chapter, we present the formal modeling framework for specifying and verifying parameterized networks of hybrid automata. The framework generalizes and unifies the modeling frameworks developed in our prior work [1–3].[1] First, we present the syntax for specifying a hybrid automaton template, and with illustrations of the expressive power and limitations of the framework. Next, we define the semantics of networks composed of (a potentially unbounded number of) instances of the template. Then, we formally define the uniform verification problem, which aims to establish properties for any number of participants in the protocol. We present methods to address the uniform verification problem in Chapters 3, 4, 5, and 6.

2.1 Introduction

Distributed systems are naturally modeled as a collection of interacting building-blocks or *modules*. For example, distributed computing systems are built from communicating computing processes, distributed traffic control protocols involve the interaction of individual vehicles, and neural networks arise from the interaction of neurons.

Hybrid systems modeling frameworks [9, 19, 23, 30, 35, 39, 40] specify state machines with combinations of discrete and continuous states and their evolution. Networks composed of hybrid automata [19, 30, 39] are useful for modeling a variety of systems, like network protocols, robotic swarms where robot positions evolve according to some ODE with control inputs determined by each robot in the swarm, and distributed vehicular traffic control protocols. State machines model discrete variables and discrete transitions, while real-valued

[1] This chapter is based in part on our prior work [1–3], portions of which are reprinted here with permission.

continuous variables evolve according to ordinary differential equations (ODEs), differential algebraic equations (DAEs), or inclusions [137].

In networks of hybrid automata, automata communicate by reading one another's state and through globally shared variables. A hybrid automaton $\mathcal{A}$ may read the variables of another hybrid automaton $\mathcal{B}$ by maintaining a *pointer* to $\mathcal{B}$. A pointer is a variable that takes values in the set of automaton identifiers or names. Pointer variables allow for modeling systems with dynamic communication topologies. Many distributed protocols utilize this type of communication, such as traffic control protocols where vehicles keep track of adjacent vehicles, swarm robotics protocols where robots keep track of neighbors, or routers that keep track of successors.

One such distributed traffic control system we present later in this chapter—see Section 2.3.8—is the Small Aircraft Transportation System (SATS) [25–27, 69]. In SATS, aircraft communicate by reading the valuations of discrete variables and continuous positions using pointers. For another example, in an automated highway system, a car may only need to keep track of the positions of the cars immediately ahead and behind it requiring two pointer variables, and similar scenarios arise in robotic swarm protocols in one-dimensional lanes [16]. However, at four-way intersections of single-lane roads, an autonomous car may need to track the positions of cars coming from every other direction, requiring three pointer variables. All of these scenarios fit into the communication model and verification framework developed in this chapter. Additionally, the framework is general enough to allow modeling of other distributed algorithms, such as Fischer's mutual exclusion algorithm, discrete mutual exclusion algorithms, and cache coherence protocols.

Outline. In Section 2.2, we informally describe the class of systems under consideration. Then in Section 2.3, we present the syntax for specifying a template hybrid automaton. The template is used as input for the *Passel* verification tool developed as a part of this dissertation, described in detail in Chapter 7. Section 2.4 describes the semantics of networks composed of copies of this template hybrid automaton, and Section 2.5 concludes the chapter.

Preliminaries. We use two symbols for referring to the number of automata in a network. Where we use N, we mean a constant, numerical natural number, that is, a fixed natural number (e.g., $\mathsf{N} = 3$). Where we use $\mathcal{N}$, we mean a symbolic natural number, that is, $\mathcal{N}$ is some arbitrary natural number. For a natural number n, we define the set $[n] \triangleq \{1, \ldots, n\}$, and we use the sets $[\mathsf{N}]$ and $[\mathcal{N}]$ for indexing automata. For a set S, we define $S_\perp \triangleq S \cup \{\perp\}$.

2.2 Informal Description of Hybrid Automata Networks

For any natural number $\mathcal{N}$ and $i \in [\mathcal{N}]$, an individual hybrid automaton $\mathcal{A}(\mathcal{N}, i)$ is a (possibly nondeterministic) state machine with finitely many discrete locations and variables of various types like reals and integers. The state of $\mathcal{A}(\mathcal{N}, i)$ can change instantaneously through discrete transitions and its real-valued variables can evolve continuously over time according to *trajectories* specified by ordinary differential equations (ODEs) or inclusions.

A network of hybrid automata $\mathcal{A}^{\mathcal{N}}$ is a collection of $\mathcal{N}$ interacting instances of a template automaton $\mathcal{A}(\mathcal{N}, i)$, in which the transitions of each hybrid automaton can depend on the the state of certain other hybrid automata. We aim to establish properties that hold for the network $\mathcal{A}^{\mathcal{N}}$ for any choice of the natural number $\mathcal{N}$. We drop the argument $\mathcal{N}$ from $\mathcal{A}(\mathcal{N}, i)$ and write $\mathcal{A}(i)$ when $\mathcal{N}$ is clear from context. In a network $\mathcal{A}^{\mathcal{N}}$, the constituent automata may communicate over discrete transitions, but not through trajectories. That is, a transition taken by $\mathcal{A}(\mathcal{N}, i)$ can depend on and influence the the state of another automaton $\mathcal{A}(\mathcal{N}, j)$, but a trajectory of $\mathcal{A}(\mathcal{N}, i)$ depends on and influences only the state of $\mathcal{A}(\mathcal{N}, i)$.

The variables of the $\mathcal{N}$ automata in the network $\mathcal{A}^{\mathcal{N}}$ are described as arrays of length $\mathcal{N}$ of appropriate types. Real-typed variable may be updated continuously and/or discretely, while variables of other types are only updated discretely. In the remainder of this chapter, we introduce the syntax for specifying networks of hybrid automata by specifying one template hybrid automaton $\mathcal{A}(\mathcal{N}, i)$, and then introduce the semantics of the language to show how networks $\mathcal{A}^{\mathcal{N}}$ composed of $\mathcal{N}$ interacting instances of the template are modeled.

2.3 Syntax for Hybrid Automaton Template $\mathcal{A}(\mathcal{N}, i)$

In this section, we define the syntax for specifying a hybrid automaton template $\mathcal{A}(\mathcal{N}, i)$ used to construct parameterized networks of hybrid automata. We begin with some preliminary definitions.

2.3.1 Variables

A *variable* is a name used for referring to state. A variable v is associated with a *type*—denoted $type(v)$—that defines a set of values the variable may take. The type of a variable may be:

(a) L: a finite set called the set of locations (defined in Section 2.3.4).

(b) $[\mathcal{N}]_\perp$: the set of automaton indices (identifiers) with the special element $\perp$ that is not equal to any index. A variable of this type is called a pointer variable or a pointer in short.

(c) $\mathbb{R}$: the set of real numbers.

(d) $\mathbb{Z}$: the set of integers.

A variable may be *local* with a name of the form $variable\_name[i]$, or *global*, in which case the name does not have the index $[i]$. For example, $q[i] : \mathsf{L}$, $p[i] : [\mathcal{N}]_\perp$, $x[i] : \mathbb{R}$, and $z[i] : \mathbb{Z}$ respectively define location, pointer, real, and integer typed local variables, while $g : [\mathcal{N}]_\perp$ is a global variable of index type. For a local variable $q[i]$, the array of variables $\langle q[1], q[2], \ldots, q[\mathcal{N}]\rangle$ is denoted by $\bar{q}^{\mathcal{N}}$. We write $\bar{q}$ when $\mathcal{N}$ is clear from context.

2.3.2 Terms, Formulas, and *Passel* Assertions

This section presents the syntax for formulas we use to specify the various syntactic components of a hybrid automaton template $\mathcal{A}(\mathcal{N}, i)$. Formulas are built-up from constants, variables, and terms of several types. Formulas are used for specifying the initial states and

the state evolution of the network. The grammar for different types of terms is as follows:

$$\begin{aligned}
\mathsf{ITerm} &::= \bot \mid 1 \mid \mathcal{N} \mid i \mid p[i] \\
\mathsf{DTerm} &::= l_c \mid q \mid q[\mathsf{ITerm}] \\
\mathsf{RTerm} &::= 0 \mid 1 \mid r_c \mid x \mid x[\mathsf{ITerm}] \\
\mathsf{ZTerm} &::= 0 \mid 1 \mid z_c \mid z \mid z[\mathsf{ITerm}]
\end{aligned}$$

An index term (ITerm) is either (a) one of the constants $\bot$, 1, $\mathcal{N}$, an index variable i, or (b) a local pointer variable p referenced at an index variable i. The grammar does not allow arbitrary productions of recursive ITerms by restricting ITerms from being produced by p[ITerm]—for example, $p[p[i]]$ is not an allowed term.[2]

Discrete terms (DTerm), real terms (RTerm), and integer terms (ZTerm) are defined as specified in the grammar. For discrete terms, l_c is constant from L, q is a discrete variable, and q[ITerm] is a discrete array referenced at an index specified by an ITerm. For real terms, r_c is a real-valued numerical constant, x is a real variable, and x[ITerm] is a real array referenced at an index specified by an ITerm. For integer terms, z_c is an integer-valued numerical constant, z is an integer variable, and z[ITerm] is an integer array referenced at an index specified by an ITerm.

Real and integer polynomials and constraints are built using the following grammar.

$$\begin{aligned}
\mathsf{RPoly} ::= {} & \mathsf{RTerm} \mid \mathsf{RPoly}_1 \ + \ \mathsf{RPoly}_2 \mid \\
& \mathsf{RPoly}_1 \ - \ \mathsf{RPoly}_2 \mid (\mathsf{RPoly}_1 \ * \ \mathsf{RPoly}_2) \\
\mathsf{RAtom} ::= {} & \mathsf{RPoly} \ < \ 0 \\
\mathsf{ZPoly} ::= {} & \mathsf{ZTerm} \mid \mathsf{ZPoly}_1 \ + \ \mathsf{ZPoly}_2 \mid \\
& \mathsf{ZPoly}_1 \ - \ \mathsf{ZPoly}_2 \mid (\mathsf{ZPoly}_1 \ * \ \mathsf{ZPoly}_2) \\
\mathsf{ZAtom} ::= {} & \mathsf{ZPoly} \ < \ 0
\end{aligned}$$

RPoly_1 and RPoly_2 (ZPoly_1 and ZPoly_2) are shorter real (integer) polynomials joined by arith-

[2]This restriction ensures that the theory is stratified [41].

metic operators—addition $+$, subtraction $-$, or multiplication $*$—to obtain longer polynomials. RAtom (ZAtom) are used for specifying real (integer) constraints. Other comparison operators—like less than or equal ($\leq$), greater than or equal ($\geq$), greater than ($>$), and equality ($=$)—will be expressed using negation ($\neg$) and conjunction ($\wedge$) in the formulas we define next.

For a polynomial p generated by RPoly (ZPoly) over n real (integer) variables $x_1, \ldots, x_n$ with k additive terms

$$p = a_1 x_1^{e_{1,1}} * \ldots * x_n^{e_{n,1}} + \ldots + a_k x_1^{e_{1,k}} * \ldots * x_n^{e_{n,k}},$$

with real (integer) coefficients $a_1, \ldots, a_k$ and natural number exponents $e_{1,1}$, ..., $e_{n,k}$, the degree of p is $deg(p) \triangleq \max_{1 \leq q \leq k}(\sum_{r=1}^{n} e_{r,q})$. If $deg(p) > 1$, then p is *nonlinear*. If $deg(p) \leq 1$, then p is *linear*. The linear fragment is the subset of formulas where all polynomials have degree at most one. We assume standard precedence of operators (e.g., $*$ before $+$, etc.).

Using these terms and constraints, formulas are defined next:

$$\mathsf{Atom} ::= \mathsf{ITerm}_1 < \mathsf{ITerm}_2 \mid \mathsf{DTerm}_1 = \mathsf{DTerm}_2 \mid \mathsf{RAtom} \mid \mathsf{ZAtom}$$

$$\mathsf{Formula} ::= \mathsf{Atom} \mid \neg\mathsf{Formula} \mid \mathsf{Formula}_1 \wedge \mathsf{Formula}_2 \mid \exists x\ \mathsf{Formula}$$

Here, x is called a *bound variable*, and is a variable of one of the types. $\mathsf{Formula}_1$ and $\mathsf{Formula}_2$ are shorter formulas that are joined by Boolean operators to obtain a longer formula. By combining the Boolean operators $\wedge$ and $\neg$ with the $<$ operator, other comparison operators, such as $=$, $\neq$, $\leq$, $>$, and $\geq$, can be expressed in formulas for indices, reals, and integers. For example, $p_1[i] = p_2[j]$ can be written as $\neg(p_1[i] < p_2[j]) \wedge \neg(p_2[j] < p_1[i])$. Universally quantified variables can be expressed by $\neg\exists x : \mathsf{Formula} \equiv \forall x : \neg\mathsf{Formula}$. Thus, we assume the language contains the standard quantifiers and Boolean operators, even if not explicitly specified by the grammar (e.g., universal quantification $\forall$, implication $\Rightarrow$, disjunction $\vee$, less-than-or-equal $\leq$, non-equality $\neq$, etc.).

If a variable in a formula is not bound, then it is called a *free variable*. If a formula does not contain any quantifiers, then it is *quantifier-free*, but otherwise it is *quantified*. If a

formula has no free variables, then it is a *sentence*. For example,

$$\forall \delta \in \mathbb{R}\ \exists x \in \mathbb{R} : x \geq \delta \text{ is a sentence, but}$$
$$\forall i \in [\mathcal{N}] : x[i] - \delta \geq 0 \text{ is not.}$$

If a formula is quantified and all the bound variables appearing in it are of index type, then it is *index-quantified*. For example,

$$\forall i \in [\mathcal{N}]\ \exists j \in [\mathcal{N}] : x[j] > x[i] \text{ is index-quantified, but}$$
$$\forall \delta \in \mathbb{R}\ \forall i \in [\mathcal{N}] : x[i] + \delta > 0 \text{ is not.}$$

An *index sentence* is a formula with no free variables of index type. For example,

$$\exists i \in [\mathcal{N}]\ \forall j \in [\mathcal{N}] : x[j] > x[i],$$
$$\forall i, j \in [\mathcal{N}] : i \neq j \Rightarrow (q[i] \neq l_c \vee q[j] \neq l_c), \text{ and}$$
$$\forall i \in [\mathcal{N}] : x[i] + \delta > 0, \text{ are index sentences, but}$$
$$x[i] + \delta > 0, \text{ and}$$
$$\forall i \in [\mathcal{N}] : (x[i] - x[j] > r_c) \vee (x[j] - x[i] > r_c) \text{ are } \textit{not} \text{ index sentences.}$$

Arithmetic operations on index terms (ITerm) are not allowed, and the allowed comparisons mean only total orders may be specified. Only equality (or non-equality) comparisons are allowed for discrete terms. If a formula is only composed of RTerms (ZTerms), then it is in the real (integer) polynomial subclass.

A formula ϕ is in disjunctive normal form (DNF) if and only if it is a disjunction of conjunctive clauses, where a conjunctive clause is one or more conjunctions of one or more atoms. A formula ϕ is in conjunctive normal form (CNF) if and only if it is a conjunction of disjunctive clauses, where a disjunctive clause is one or more disjunctions of one or more atoms.

The *Passel assertion language* is the set of index-quantified formulas generated by the grammar just defined. For a formula ϕ, let $vars(\phi)$ be the set of variables appearing in

ϕ. For a formula ϕ, let *ivars*(ϕ) be the set of distinct index variables appearing in ϕ. For a formula ϕ, let *free*(ϕ) be the set of free variables appearing in ϕ. For a formula ϕ, let *bound*(ϕ) be the set of bound variables appearing in ϕ. For a quantified formula ϕ, let *body*(ϕ) be the body of the quantifier, with all bound variables *bound*(ϕ) replaced with universally instantiated variables with the same names. For a set of variable names V and a formula ϕ, if *free*(ϕ) $\subseteq \mathsf{V}$, then ϕ is *over* V. For a set of variable names V and a formula ϕ, if *free*(ϕ) $= \mathsf{V}$, then ϕ is *over all* V. For example, for the set of variable names $\mathsf{V} \triangleq \{i, j, x[i], q[i], g\}$—where *type*($i$) $= [\mathcal{N}]$, *type*(j) $= [\mathcal{N}]$, *type*($x[i]$) $= \mathbb{R}$, *type*($q[i]$) $= \mathsf{L}$, and *type*(g) $= [\mathcal{N}]_\perp$—the following specify:

$g = \perp$,

$\forall i \in [\mathcal{N}] : q[i] = \mathsf{cs} \Rightarrow x[i] \geq 0$, and

$\forall i, j \in [\mathcal{N}] : q[i] = \mathsf{cs} \wedge q[j] = \mathsf{cs} \Rightarrow x[i] \geq L_S \wedge x[j] \geq 0$ formulas over V, but

$i \neq j \Rightarrow x[j] > x[i]$ and

$i \neq j \Rightarrow p[i] = j$, are not formulas over V.

Here, $\perp$, L_S, and cs are constants. For a *Passel* assertion over a set of variables V, we always assume that a countable set of symbolic automaton indices are included in V for referencing different variables. *Passel* assertions over particular sets of variables—along with further restrictions, such as being quantifier-free—will be used for specifying various syntactic components of the hybrid automaton template $\mathcal{A}(\mathcal{N}, i)$.

2.3.3 Hybrid Automaton Template

We next define a syntactic structure called a hybrid automaton template, which we use to specify the behavior of a participant in a parameterized network.

Definition 2.1 *For symbolic constants $\mathcal{N} \in \mathbb{N}$ and $i \in [\mathcal{N}]$, a* hybrid automaton template *$\mathcal{A}(\mathcal{N}, i)$ is specified by the following syntactic components:*

(a) V_i: a finite set of variable names,

(b) L*: a finite set of location names,*

(c) Init_i*: an initial condition, which is a Passel assertion over* V_i*,*

(d) Trans_i*: a finite set of discrete transition statements, each of which is composed of a from-to pair of locations, along with a guard, a universal guard, and an effect, which are quantifier-free Passel assertions over* $\mathsf{V}_i \cup \mathsf{V}'_i$*, where* V'_i *is the set of primed variable names corresponding to* V_i*, and*

(e) Flow_i*: a finite set of trajectory statements, one for each element in* L*, each of which is composed of an invariant, a stopping condition, and a flowrate, each of which are quantifier-free Passel assertions over* $\mathsf{V}_i \cup \mathsf{V}_i\_dot$*, where* V_i_dot *is the set of dotted variable names corresponding to the real-valued variables in* V_i*.*

The subscript i emphasizes that components may use the automaton's index.

A hybrid automaton template $\mathcal{A}(\mathcal{N}, i)$ is written $\mathcal{A}(i)$ when $\mathcal{N}$ is clear from context. Throughout this section, we use an example specification of Fischer's mutual exclusion protocol Fischer to illustrate the language constructs available for specifying a hybrid automaton template $\mathcal{A}(i)$. The specification of the protocol in this language[3] is shown in Figure 2.1, and an equivalent graphical representation appears in Figure 2.2.

2.3.4 Specifying Locations

The set of location names L is specified by a list of location names. A location name follows the keyword `location name`. In Fischer, the set of locations L is $\{\mathsf{rem}, \mathsf{try}, \mathsf{wait}, \mathsf{cs}\}$ (lines 11, 13, 17, and 19). Locations are depicted graphically as the circles in Figure 2.2 for Fischer. Each automaton $\mathcal{A}(i)$ has a single local variable $q[i]$ that takes values in L. A trajectory statement may follow each location name, defined in detail in Section 2.3.7. Locations are specified in this manner—instead of only using a variable of type L—to allow for easily specifying continuous dynamics varying from one location to another.

[3]Formulas are as specified in Section 2.3.2, except quantifiers and operators with their text (parsing) equivalents. For example, $\wedge$ is `and`, $\vee$ is `or`, $\forall$ is `forall`, $\exists$ is `exists`, $\leq$ is `<=`, $\geq$ is `>=`, $=$ is `=`, $\neq$ is `!=`, $\Rightarrow$ is `implies`, etc. Figure 2.1 is marked-up for readability, but is essentially in *Passel*'s input language.

```
1  parameter name='A' type='real' value = 5.0  // smaller timing parameter
   parameter name='B' type='real' value = 35.0 // larger timing parameter
3  parameter name='lb' type='real' value = 1.0 // lower clock rate
   parameter name='ub' type='real' value = 2.0 // upper clock rate
5
   automaton name='Fischer'
7    variable name='q[i]' type='L' // control location local variable
     variable name='x[i]' type='real' // continuous local variable
9    variable name='g' type='index' // global lock variable

11   location name='rem'
       flowrate: x[i]_dot = 0.0
13   location name='try'
       inv: x[i] <= A
15     stop: x[i] = A
       flowrate: x[i]_dot >= lb and x[i]_dot <= ub
17   location name='wait'
       flowrate: x[i]_dot >= lb and x[i]_dot <= ub
19   location name='cs'
       flowrate: x[i]_dot = 0.0
21
     transition from='rem' to='try'
23     grd: g = ⊥
       eff: x[i]′ = 0.0
25   transition from='try' to='wait'
       eff: g′ = i and x[i]′ = 0.0
27   transition from='wait' to='cs'
       grd: g = i and x[i] >= B
29     eff: x[i]′ = 0.0
     transition from='wait' to='rem'
31     grd: g != i and x[i] >= B
       eff: x[i]′ = 0.0
33   transition from='cs' to='rem'
       eff: g′ = ⊥ and x[i]′ = 0.0
35
   property: forall i j ((i != j and q[i] = cs) implies (q[j] != cs))
37 initially: forall i (q[i] = rem and x[i] = 0 and g = ⊥)
```

Figure 2.1: Hybrid automaton template specifying $\mathcal{A}(\mathcal{N}, i)$ for Fischer's mutual exclusion algorithm Fischer, which is also used as the input to *Passel*.

2.3.5 Specifying Variables, Parameters, Initial Conditions, and Invariant Properties

The set of variables V_i is specified by the list of variable names and types following the keywords `variable name` and `type`. For the Fischer automaton with index i (Figure 2.1), the set of variables is specified by the list of variables on lines lines 7 through 9. It has two local variables, $q[i]$ and $x[i]$, with types L and $\mathbb{R}$, and a single global variable g of type $[\mathcal{N}]_\perp$.

The specification of $\mathcal{A}(i)$ may use a set of symbolic or numerical parameters (constants). Each parameter is specified by its name, type, and, optionally, a quantifier-free *Passel* assertion that specifies constraints that the parameters must satisfy. For the Fischer example, there are four real-valued parameters, A, B, lb, and ub (lines 1, 2, 3, and 4).

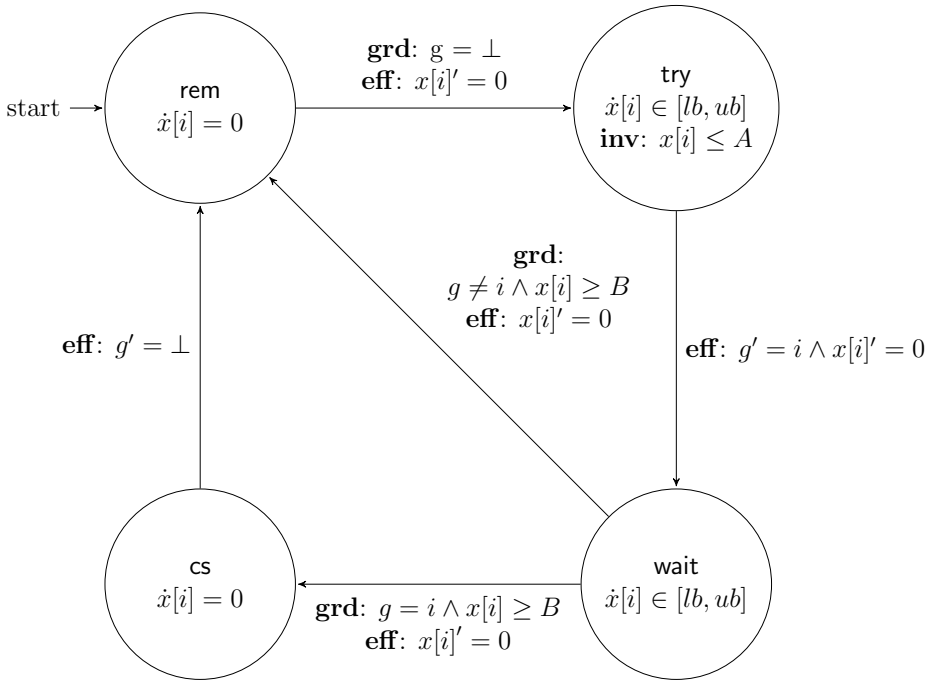

Figure 2.2: Graphical depiction of the hybrid automaton template $\mathcal{A}(\mathcal{N}, i)$ from Figure 2.1 specifying Fischer's mutual exclusion algorithm Fischer.

We denote the set of local variables by $\mathsf{V}_L[i]$, the set of global variables by $\mathsf{V}_G[i]$, and the set of parameters by $\mathsf{V}_P[i]$. In the Fischer example, $\mathsf{V}_L[i] = \{q[i], x[i]\}$, $\mathsf{V}_G[i] = \{g\}$, and $\mathsf{V}_P[i] = \{A, B, lb, ub\}$. When clear from context, we drop the index i and write V_L, V_G, and V_P for $\mathsf{V}_L[i]$, $\mathsf{V}_G[i]$, and $\mathsf{V}_P[i]$, respectively.

Initial Conditions. The initial condition assertion Init_i is a universally index-quantified *Passel* assertion following the keyword `initially`. In Fischer, the initial condition assertion is (line 37):

$$\text{forall i} \ : \ (q[i] = \mathsf{rem} \text{ and } x[i] = 0 \text{ and } g \ = \bot),$$

where i is implicitly quantified over $[\mathcal{N}]$. The initial condition assertion for Fischer asserts that, for each index $i \in [\mathcal{N}]$, the variables of $\mathcal{A}(i)$ have the constraints $q[i] = \mathsf{rem}$ and $x[i] = 0$, and that the global variable $g = \bot$. If a variable $v \in \mathsf{V}_i$ is not specified in the initial condition, it is assumed that v is initially an arbitrary value in its type $type(v)$. Note that the Init_i assertion may specify constraints over all automata in the network using universally

index-quantified *Passel* assertions.

Candidate Invariant Properties. Candidate invariant properties are specified as *Passel* assertions following the keyword `property`. For example, a mutual exclusion invariant property can be specified as (line 36):

$$\text{forall i, j} \ : \ ((i \ != j \text{ and } q[i] = \mathsf{cs}) \text{ implies } (q[j] \ != \mathsf{cs})),$$

where i and j are implicitly quantified over the set of indices $[\mathcal{N}]$.

2.3.6 Specifying Discrete Transitions

For any $\mathcal{N} \in \mathbb{N}$ and any $i \in [\mathcal{N}]$, the set of discrete transitions Trans_i is specified by the list of transition statements following the keyword `transition`. Each transition statement specifies a from-to pair of locations following the keywords `from` and `to`. There is at most one transition statement between each pair of locations. If it exists, we will denote a transition from location src to location dest by $\mathsf{t}(\mathsf{src}, \mathsf{dest}) \in \mathsf{Trans}_i$, which is written as t when the from-to locations are clear from context.

Each transition $\mathsf{t} \in \mathsf{Trans}_i$ *may* specify a *guard* following the keyword **grd**, a *universal guard* following the keyword **ugrd**, and an *effect* following the keyword **eff**. The guard, universal guard, and effect are quantifier-free *Passel* assertions, and they are denoted by $\mathbf{grd}(\mathsf{t}, i)$, $\mathbf{ugrd}(\mathsf{t}, i)$, and $\mathbf{eff}(\mathsf{t}, i)$ for $\mathcal{A}(i)$, respectively. If i is clear from context, we drop it and write $\mathbf{grd}(\mathsf{t})$, $\mathbf{ugrd}(\mathsf{t})$, or $\mathbf{eff}(\mathsf{t})$.

The universal guard is a quantifier-free *Passel* assertion involving the variables V_j, for $j \neq i$, and we recall i is the index of the template $\mathcal{A}(\mathcal{N}, i)$. The universal guard specifies an assertion over the variables of other automata and global variables. Such assertions over the variables of all the other automata in the network are useful for modeling broadcast-like communications.

The effect models the update of state, and is a quantifier-free *Passel* assertion over the variables $\mathsf{V}_i \cup \mathsf{V}'_i$, where V'_i concatenates a prime ($\prime$) to each variable name v in V_i. The effect specifies a relation between the variables before (unprimed) and after (primed) the transition.

The effect is not required to specify variables that are not modified by the transition.

The transition from wait to cs for Fischer with index i (line 28), where

$$\textbf{grd} : g = i \text{ and } x[i] >= B,$$
$$\textbf{eff} : x[i]' = 0.0$$

specifies that automaton i *may* nondeterministically transition from a state where $q[i] = \mathsf{wait}$ to a state where $q[i]' = \mathsf{cs}$, only if the global pointer variable g is equal to i and the local real-valued variable $x[i]$ is at least as large as the parameter value B. Further, if the automaton *does* make this transition, then the effect specifies that $x[i]$ is to be reset to 0. If the guard condition is omitted, then it is assumed to be just the control location condition. For example, in Fischer, the transition from try to wait is enabled when $q[i] = \mathsf{try}$.

2.3.7 Specifying Continuous Dynamics

The elements of the set of trajectory statements Flow_i are listed following the corresponding location names. Each location $\mathsf{m} \in \mathsf{L}$ has a trajectory statement in Flow_i. The trajectory statement consists of an invariant condition following the keyword **inv**, a stopping condition following the keyword **stop**, and a sequence of flow rates following the keyword **flowrate**. The invariant condition of a location $\mathsf{m} \in \mathsf{L}$ for automaton $\mathcal{A}(i)$ is denoted by $\textbf{inv}(\mathsf{m}, i)$, the stopping condition is denoted by $\textbf{stop}(\mathsf{m}, i)$, and the flow rate for some real-valued variable $x[i] \in \mathsf{V}_i$, is denoted by $\textbf{flowrate}(\mathsf{m}, x[i])$.

The invariant and stopping conditions are *Passel* assertions involving only the real variables $\mathsf{X}[i]$ and real parameters in $\mathsf{V}_P[i]$. The flow rate associates each real-valued variable in $\mathsf{X}[i]$ with ordinary differential equations or inclusions specified as the real polynomial subclass of *Passel* assertions. The flow rate for a variable name $x[i] \in \mathsf{V}_i$ is specified by concatenating *_dot* to the variable name, for example $x[i]$*_dot*. For example, the trajectory statement for a two-dimensional linear differential equation with an invariant and stopping condition is

specified as:

$$\begin{aligned}
&\textbf{inv}: && x_1[i] <= c_1 \text{ and } x_2[i] <= c_2 \\
&\textbf{stop}: && x_1[i] = s_1 \text{ or } x_2[i] = s_2 \\
&\textbf{flowrate}: && x_1[i]_\text{dot} = a_{11} * x_1[i] + a_{12} * x_2[i] + b_1 \\
& && x_2[i]_\text{dot} = a_{21} * x_1[i] + a_{22} * x_2[i] + b_2,
\end{aligned} \tag{2.1}$$

where c_1, c_2, s_1, s_2, a_{11}, a_{12}, a_{21}, a_{22}, b_1, and b_2 are real parameters. This specifies the following ODE in matrix form,

$$\begin{bmatrix} \dot{x_1}[i] \\ \\ \dot{x_2}[i] \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1[i] \\ \\ x_2[i] \end{bmatrix} + \begin{bmatrix} b_1 \\ \\ b_2 \end{bmatrix}. \tag{2.2}$$

The following example specifies the special case of *rectangular differential inclusions*, where the time-derivative is specified by an upper and a lower bound in terms of a numerical constant or a parameter name:

$$\begin{aligned}
&\textbf{flowrate}: && x_1[i]_\text{dot} >= lb_1 \text{ and } x_1[i]_\text{dot} <= ub_1 \\
& && x_2[i]_\text{dot} >= lb_2 \text{ and } x_2[i]_\text{dot} <= ub_2,
\end{aligned}$$

where $lb_1 \leq ub_1$ and $lb_2 \leq ub_2$ are real parameters or numerical values. This yields the following rectangular differential inclusions on $\dot{x_1}[i]$ and $\dot{x_2}[i]$:

$$\begin{bmatrix} \dot{x_1}[i] \\ \\ \dot{x_2}[i] \end{bmatrix} \in \begin{bmatrix} [lb_1, ub_1] \\ \\ [lb_2, ub_2] \end{bmatrix}. \tag{2.3}$$

2.3.8 Example: Simple Small Aircraft Transportation System (SATS) Landing Protocol

An essential part of the Small Aircraft Transportation System (SATS) [2, 25–27, 69]—a distributed air traffic control protocol we define completely in Chapter 3, Section 3.2—is used as another example illustrating the specification language of a template automaton $\mathcal{A}(\mathcal{N}, i)$. SATS was designed to increase throughput at small airports without air traffic controllers by allowing aircraft to coordinate among themselves with minimal assistance from a centralized communication component [25, 26, 138]. Aircraft in SATS communicate by reading the continuous position of any aircraft immediately ahead of it in the landing sequence, where the aircraft immediately ahead is tracked using its index.

The hybrid automaton template specifying the essential behavior of SATS is shown in Figure 2.3, and its properties are shown in Figure 2.4. SATS is depicted graphically in Figure 1.3. SATS is parameterized on the number of aircraft involved in the landing attempt, so each aircraft is naturally specified as a hybrid automaton template $\mathcal{A}(\mathcal{N}, i)$. The network $\mathcal{A}^{\mathcal{N}}$ models a single airport and $\mathcal{N}$ flying aircraft that are attempting to land. After determining the landing sequence order from a centralized airport management module (AMM)—which is modeled with a global variable $last$—the remainder of the protocol is decentralized and each aircraft communicates with the aircraft immediately ahead of it (if one exists) to determine if it is safe to attempt landing.

All aircraft begin in the fly location, and when an aircraft is ready to attempt landing, it initiates the approach to the airport by making a transition to the hold location. The hold location physically represents that the aircraft is flying in a cyclic holding pattern, and it is assumed that the aircraft maintain a safe separation in this location. On entering hold, an aircraft is either designated as the first one in the landing sequence (and $next[i]' = \perp$), or the aircraft is assigned the index of the last aircraft that began its approach to the runway (and $next[i]' = last$). Subsequently, an aircraft may nondeterministically transition from the hold location to the base location which represents that it is physically approaching the runway.

The position of the i^{th} aircraft is modeled using a single continuous variable ($x[i]$) of real type, representing the position along a line measured starting from the geographic location of the cyclic holding zone (that is, the beginning of the base region). This transition is only

```
1  parameter name='L_B' type='real' value = 120.0  // base zone length
   parameter name='L_S' type='real' value = 5.0    // separation spacing
3  parameter name='v_min' type='real' value = 1.0  // minimum velocity
   parameter name='v_max' type='real' value = 2.0  // maximum velocity
5
   automaton name='SSATS'
7    variable name='q[i]' type='L'          // location local variable
     variable name='next[i]' type='index'   // next aircraft (if any)
9    variable name='x[i]' type='real'       // continuous local variable
     variable name='last' type='index'      // global lock variable
11
     location name='fly'
13     flowrate: x[i]_dot = 0.0
     location name='hold'
15     flowrate: x[i]_dot = 0.0
     location name='base'
17     inv: x[i] <= L_B
       stop: x[i] = L_B
19     flowrate: x[i]_dot >= v_min and x[i]_dot <= v_max
     location name='runway'
21     flowrate: x[i]_dot = 0

23   transition from='fly' to='hold'
       eff: next[i]' = last and x[i]' = 0.0 and last' = i
25
     transition from='hold' to='base'
27     grd: next[i] != ⊥ implies (q[next[i]] = base and x[next[i]] >= L_S)
       eff: x[i]' = 0.0
29
     transition from='base' to='hold'
31     grd: x[i] >= L_B and next[i] = ⊥
       eff: x[i]' = 0 and (last != i implies next'[i] = last) and last' = i
33     ugrd: next[j] = i implies next[j]' = ⊥

35   transition from='base' to='runway'
       grd: x[i] >= L_B and next[i] = ⊥
37     eff: (last = i implies last' = ⊥)
       ugrd: next[j] = i implies next[j]' = ⊥
39
   initially: forall i (q[i] = fly and next[i] = ⊥ and last = ⊥)
```

Figure 2.3: *Passel* input file specifying hybrid automaton $\mathcal{A}(i)$ for SSATS, a simplified SATS protocol.

enabled for aircraft i if there is at least L_S distance between its position $x[i]$ and the position of the aircraft ahead of it, $x[next[i]]$ (if one exists). Once in the base location, the aircraft is approaching the runway and after traversing L_B distance, the aircraft may either (a) cancel the landing attempt and return to the cyclic holding pattern in location hold, in which case it becomes the last aircraft in the sequence, or (b) the aircraft may succeed in landing and set its location to runway.

We use *Passel* to automatically prove the properties in lines 1 through 7 for SSATS, with memory and time required as presented in Chapter 7. These properties represent an inductive invariance proof of safe separation—that aircraft are always at least L_S distance

```
   property: forall i q[i] = fly implies last != i
2  property: forall i, j next[j] = i implies  q[i] != fly
   property: forall i, j (q[i] = hold and next[j] = i) implies q[j] = hold
4  property: forall i, j (q[i] = base and q[j] = base and next[j] = i )
          implies x[i] >= L_S + (v_max - v_min)(L_B  - x[j]) / v_min
6  property: forall i, j ( i != j and q[i] = base and q[j] = base and
          next[j] = i ) implies x[i] - x[j] >= L_S
```

Figure 2.4: *Passel* input file specifying properties for SSATS, a simplified SATS protocol.

apart, which we define formally in Section 2.4.

2.4 Semantics of Hybrid Automata Networks

The semantics of the hybrid automata network $\mathcal{A}^{\mathcal{N}}$—where an arbitrary number $\mathcal{N} \in \mathbb{N}$ of instances of the template $\mathcal{A}(\mathcal{N}, i)$ operate in parallel—is defined in this section.

2.4.1 Parameterized Network of Hybrid Automata

The semantics are defined in terms of a transition system with a set of variables $\mathsf{V}^{\mathcal{N}}$, a set of states $Q^{\mathcal{N}}$, a set of initial states $\Theta^{\mathcal{N}}$, and a transition relation $T^{\mathcal{N}}$. For networks of hybrid automata, none of these sets is usually finite since variables may have real types.

Definition 2.2 (Parameterized Network of Hybrid Automata) *For any $\mathcal{N} \in \mathbb{N}$, a* parameterized network of hybrid automata *is a tuple $\mathcal{A}^{\mathcal{N}} \triangleq \langle \mathsf{V}^{\mathcal{N}}, Q^{\mathcal{N}}, \Theta^{\mathcal{N}}, T^{\mathcal{N}} \rangle$, where*

(a) $\mathsf{V}^{\mathcal{N}}$ are the variables of the network,

$$\mathsf{V}^{\mathcal{N}} \triangleq \mathsf{V}_G \cup \bigcup_{i=1}^{\mathcal{N}} \mathsf{V}_L[i],$$

(b) $Q^{\mathcal{N}} \subseteq val(\mathsf{V}^{\mathcal{N}})$ is the state-space,

(c) $\Theta^{\mathcal{N}} \subseteq Q^{\mathcal{N}}$ is the set of initial states, and

(d) $T^{\mathcal{N}} \subseteq Q^{\mathcal{N}} \times Q^{\mathcal{N}}$ is the transition relation, which is partitioned into disjoint sets of discrete transitions $\mathcal{D}^{\mathcal{N}} \subseteq Q^{\mathcal{N}} \times Q^{\mathcal{N}}$ and continuous trajectories $\mathcal{T}^{\mathcal{N}} \subseteq Q^{\mathcal{N}} \times Q^{\mathcal{N}}$.

The transition system is said to be parameterized on $\mathcal{N}$ since fixing different values of $\mathcal{N}$ yield different transition systems. This definition allows for proving an invariant property ζ for *every* network of hybrid automata. For instance $\forall \mathcal{N} \in \mathbb{N} : \left(\mathcal{A}^{\mathcal{N}} \models \zeta(\mathcal{N})\right)$ states that, for every choice of $\mathcal{N} \in \mathbb{N}$, the corresponding network of hybrid automata $\mathcal{A}^{\mathcal{N}}$ satisfies the property $\zeta(\mathcal{N})$.

2.4.2 State-Space and Semantics of *Passel* Assertions

Recall that V_i is the set of variable names for the hybrid automaton template $\mathcal{A}(\mathcal{N}, i)$ where $i \in [\mathcal{N}]$. A state $\mathbf{x}$ in $Q^{\mathcal{N}}$ of $\mathcal{A}^{\mathcal{N}}$ is defined in terms of the valuations of all the variables of all its components. For each $i \in [\mathcal{N}]$, the *valuation* of a variable $v \in \mathsf{V}_i$ is a function that associates the variable name v to a value in its type $\mathit{type}(v)$. Elements of the state-space $Q^{\mathcal{N}}$ are called *states* and are denoted by boldface $\mathbf{v}$, $\mathbf{v}'$, etc.

At a state $\mathbf{v}$, the valuation of a particular local variable $x[i] \in \mathsf{V}_L[i]$ for automaton $\mathcal{A}(i)$ is denoted by $\mathbf{v}.x[i]$, and $\mathbf{v}.g$ for some global variable g in $\mathsf{V}_G[i]$. We recall that we refer to the valuations of a local variable $v[i] \in \mathsf{V}_L$ of all $\mathcal{N}$ automata in the network $\mathcal{A}^{\mathcal{N}}$ as an array variable, and denote it $\bar{v}$ which takes values in $\mathit{type}(v[i])^{\mathcal{N}}$. So, for a state $\mathbf{v}$, the valuations of a local variable $v[i] \in \mathsf{V}_L[i]$ for all $i \in \mathcal{N}$ is written $\mathbf{v}.\bar{v}$. The valuation of all the local variables for automaton $\mathcal{A}(i)$ at state $\mathbf{v}$ is denoted by $\mathbf{v}.\mathsf{V}_L[i]$. The valuation of all the local variables for automaton $\mathcal{A}(i)$, as well as the global variables, at state $\mathbf{v}$ is denoted by $\mathbf{v}.\mathsf{V}_i$. The state-space Q_i corresponding to automaton $\mathcal{A}(\mathcal{N}, i)$ in the network $\mathcal{A}^{\mathcal{N}}$ is defined as $Q_i \triangleq \mathit{val}(\mathsf{V}_i)$.

Representing States with Assertions. Subsets of $Q^{\mathcal{N}}$ are often represented by assertions involving the variables. If a state $\mathbf{v}$ satisfies a formula ϕ—that is, the corresponding variable valuations result in ϕ evaluating to *true*—we write $\mathbf{v} \models \phi$. For such a formula ϕ, the corresponding states satisfying ϕ are denoted by $[\![\phi]\!]$. A *model* for an assertion provides interpretation to the elements appearing in the assertion.

Definition 2.3 *An* n-model M *for an assertion* ψ *is denoted* $M(n, \psi)$ *and provides an interpretation of each the free variables in* ψ *as follows:*

- *the index constants* $\perp$, 1, *and* $\mathcal{N}$ *are respectively assigned the values* 0, 1, *and* n,
- *each pointer variable is assigned a value in the set* $[n]$,
- *each discrete variable is assigned a value in* L,
- *each real variable is assigned a value in* $\mathbb{R}$,
- *each integer variable is assigned a value in* $\mathbb{Z}$, *and*
- *each pointer, discrete, real, and integer array is assigned respectively a* $\{0, \ldots, n\}$*-valued,* L*-valued, real-valued, or integer-valued array of length* n.

Given an assertion ψ and a model $M(n, \psi)$, if ψ evaluates to true with the interpretations of the free variables given by $M(n, \psi)$, then $M(n, \psi)$ is said to *satisfy* ψ. If there exist models that satisfy ψ, then the assertion is said to be *satisfiable*. If all models of ψ satisfy it, then the assertion is said to be *valid*.

Initial States. The set of initial states $\Theta^{\mathcal{N}} \subseteq Q^{\mathcal{N}}$ is defined as $[\![\mathsf{Init}_i]\!]$, that is, the set of states satisfying the *Passel* assertion Init_i:

$$\Theta^{\mathcal{N}} \triangleq \{\mathbf{v} \in Q^{\mathcal{N}} \mid \mathbf{v} \models \mathsf{Init}_i\}.$$

In Fischer (Figure 2.1), the set of initial states specified by line 37 is

$$\begin{aligned}\Theta^{\mathcal{N}} \triangleq [\![\mathsf{Init}_i]\!] &= \{\mathbf{x} \in Q^{\mathcal{N}} \mid \forall i \in [\mathcal{N}], \mathbf{x}.q[i] = \mathsf{rem} \wedge \mathbf{x}.x[i] = 0.0 \wedge g = \perp\} \\ &= \{\mathbf{x} \in Q^{\mathcal{N}} \mid \mathbf{x} \models \forall i \in [\mathcal{N}], q[i] = \mathsf{rem} \wedge x[i] = 0.0 \wedge g = \perp\},\end{aligned}$$

where we have indicated equivalent ways of writing the set of initial states using the notations introduced earlier in this section, each of which are useful in different contexts.

2.4.3 Transitions and Trajectories

The evolution of the states of $\mathcal{A}^{\mathcal{N}}$ are describing by a transition relation $T^{\mathcal{N}} \subseteq Q^{\mathcal{N}} \times Q^{\mathcal{N}}$. For a pair $(\mathbf{v}, \mathbf{v}') \in T^{\mathcal{N}}$, we use the notation $\mathbf{v} \to \mathbf{v}'$, where $\mathbf{v}$ is called the *pre-state* and $\mathbf{v}'$

is called the *post-state*. There are two ways state is updated by $T^{\mathcal{N}}$: discrete transitions $\mathcal{D}^{\mathcal{N}}$ model instantaneous change of state and continuous trajectories $\mathcal{T}^{\mathcal{N}}$ model change of state after a time interval.

Discrete Transitions. Discrete transitions model atomic, instantaneous updates of state due to *one* automaton in the network $\mathcal{A}^{\mathcal{N}}$. There is a discrete transition $\mathbf{v} \to \mathbf{v}' \in \mathcal{D}^{\mathcal{N}}$ iff:

$$\begin{aligned}\exists i \in [\mathcal{N}]\ \exists \mathsf{t} \in \mathsf{Trans}_i : \mathbf{v}.\mathsf{V}_i \models \mathbf{grd}(\mathsf{t}, i) \wedge \mathbf{v}'.\mathsf{V}_i \models \mathbf{eff}(\mathsf{t}, i) \wedge \\ (\forall j \in [\mathcal{N}] : \mathbf{v}.\mathsf{V}_j \models \mathbf{ugrd}(\mathsf{t}, j) \wedge j \neq i \Rightarrow \mathbf{v}'.\mathsf{V}_j = \mathbf{v}.\mathsf{V}_j).\end{aligned}$$

From the pre-state $\mathbf{v}$, any automaton in the network, with any transition satisfying the guard *may* update its post-state according to transition effect, while the states of the other automata remain unchanged. Informally, a discrete transition from pre-state $\mathbf{v}$ to post-state $\mathbf{v}'$ models the discrete transition of one particular hybrid automaton $\mathcal{A}(i)$ by some transition $\mathsf{t} \in \mathsf{Trans}_i$. The universal guard of transition t depends on the variables in V_j for $j \neq i$.

We recall that the guard is a quantifier-free *Passel* assertion and specifies the enabling condition for the transition, which is a condition that must evaluate to true to allow the transition to update the system state. If the guard or universal guard are not specified, we assume they are *true*, which means a transition $\mathsf{t}(\mathsf{src}, \mathsf{dest})$ may only be taken by automaton $\mathcal{A}(i)$ if $q[i] = \mathsf{src}$. We assume the identity relation for any primed variable $v' \in \mathsf{V}_i'$ not specified in an effect. For example, if $x[i]'$ is not specified in an effect, then we assume the specified effect is conjuncted with $x[i]' = x[i]$.

For the Fischer example, the semantics for the discrete transition $\mathsf{t}(\mathsf{wait}, \mathsf{cs})$ (line 27) for some automaton $\mathcal{A}(i)$ are defined by:

$$\begin{aligned}(\mathbf{v}, \mathbf{v}') \models \exists i \in [\mathcal{N}]\ &: (q[i] = \mathsf{wait} \wedge x[i] \geq B \wedge g = i) \wedge \\ &\quad (q[i]' = \mathsf{cs} \wedge x[i]' = 0.0 \wedge g' = g) \wedge \\ (\forall j \in [\mathcal{N}]\ &: j \neq i \Rightarrow q[j]' = q[j] \wedge x[j]' = x[j]).\end{aligned}$$

This transition can occur from states $\mathbf{v}$ where $\mathbf{v} \models q[i] = \mathsf{wait} \wedge x[i] \geq B \wedge g = i$. This

transition updates the location of $\mathcal{A}(i)$ to be $q[i]' = \mathsf{cs}$, the real variable $x[i]' = 0$, and does not change the global variable g, and additionally, the variables of no other automata $\mathcal{A}(j)$ are updated. Together, this is defined by all states $\mathbf{v}' \models q[i]' = \mathsf{cs} \wedge x[i]' = 0.0 \wedge g' = g \wedge (\forall j \in [\mathcal{N}] \ : \ j \neq i \Rightarrow q[j]' = q[j] \wedge x[j]' = x[j])$.

Continuous Trajectories. Continuous trajectories model update of state over intervals of time. There is a trajectory $\mathbf{v} \to \mathbf{v}' \in \mathcal{T}^{\mathcal{N}}$ iff some amount of time—t_e—can elapse from $\mathbf{v}$, such that,

(a) the states of *all* automata in the network $\mathcal{A}^{\mathcal{N}}$ are updated to $\mathbf{v}'$ according to their individual trajectory statements,

(b) while ensuring the invariants of *all* automata along the entire trajectory, and

(c) that if the stopping condition of *any* automaton is satisfied, it is at the end of a trajectory.

Formally, trajectories are defined as solutions of differential equations or inclusions specified in the trajectory statements of $\mathcal{A}(i)$. The differential equation $\dot{x} = f(x)$ where $x \in \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}^n$ has a solution for initial condition $x_0 \in \mathbb{R}^n$ if there exists a differentiable function $\gamma(t)$ for $\gamma : \mathbb{R}_{\geq 0} \to \mathbb{R}^n$ such that $\gamma(0) = x_0$ and, for every $\tau \in [0, t]$, $\dot{\gamma}(\tau) = f(\gamma(\tau))$. A differential inclusion is $\dot{x} \in F(x)$ for $x \in \mathbb{R}^n$, where F is a set-valued function from $\mathbb{R}^n$ to $\mathbb{R}^n$, so that $F(x) \subseteq \mathbb{R}^n$. A solution for the differential inclusion with initial condition x_0 is any differentiable function $\gamma(t)$ for $\gamma : \mathbb{R}_{\geq 0} \to \mathbb{R}^n$ such that $\gamma(0) = x_0$ and, for every $\tau \in [0, t]$, $\dot{\gamma}(\tau) \in F(\gamma(\tau))$. Sufficiently smooth differential equations satisfying continuity conditions—such as Lipschitz continuity [139]—have unique solutions, whereas differential inclusions have families of solutions [111, 137].

Thus, to define trajectories for $\mathcal{A}^{\mathcal{N}}$ formally, we first define a set-valued function called $\mathbf{flow}(\mathsf{m}, \mathbf{v}.\mathsf{V}_L[i], t)$ that returns the states of $\mathcal{A}(i)$ when $q[i] = \mathsf{m}$ that can be reached from $\mathbf{v}.\mathsf{V}_L[i]$ in t time. We suppose $\mathbf{flow}(\mathsf{m}, \mathbf{v}.\mathsf{V}_L[i])$ is set-valued, as this subsumes the case when $\mathbf{flowrate}(\mathsf{m}, x[i])$ specifies a differential equation for $\dot{x}[i]$ instead of an inclusion.

Let $n = |\mathsf{X}[i]|$ be the number of continuous local variables in the template $\mathcal{A}(i)$. Let $[\![\mathbf{flowrate}(\mathsf{m}, \mathsf{X}[i])]\!] : \mathbb{R}^n \to \mathbb{R}^n$ be the vector of differential inclusions (and/or equations)

for all the continuous variables $\mathsf{X}[i]$ of $\mathcal{A}(i)$, assumed to be ordered lexicographically by the variable names. For example, for Fischer in wait, $[\![\mathbf{flowrate}(\mathsf{wait}, \mathsf{X}[i])]\!]$ is $lb \leq \dot{x}[i] \leq ub$ (Figure 2.1, line 18) since there is a single continuous variable $x[i]$ specified to evolve according to the rectangular differential inclusion with lower and upper bounds lb and ub, respectively.

Here, $\mathbf{v}'.\mathsf{V}_L[i] = \mathbf{flow}(\mathsf{m}, \mathbf{v}.\mathsf{V}_L[i], t)$ iff[4]

(a) for each real local variable $x[i] \in \mathsf{X}[i]$, $\dot{x}[i]$ with initial condition $\mathbf{v}.x[i]$ has a solution $\gamma(t)$ $\mathbf{v}'.x[i] = \gamma(t)$, and

(b) for each non-real local variable $y[i] \in \mathsf{V}_L[i] \setminus \mathsf{X}[i]$, $\mathbf{v}'.y[i] = \mathbf{v}.y[i]$.

For the Fischer example for wait (Figure 2.1, line 17),[5]

$$\begin{aligned}\mathbf{flow}(\mathsf{wait}, \mathbf{v}.x[i], t) &\in \mathbf{v}.x[i] + [lb * t, ub * t] \\ &\in [\mathbf{v}.x[i] + lb * t, \mathbf{v}.x[i] + ub * t], \text{equivalently,} \\ \mathbf{v}.x[i] + lb * t \leq &\mathbf{v}'.x[i] \leq \mathbf{v}.x[i] + ub * t.\end{aligned}$$

Thus far, we have not included the invariant and stopping condition, so we include these to complete the definition of trajectories. There is a trajectory $\mathbf{v} \to \mathbf{v}' \in \mathcal{T}^{\mathcal{N}}$ iff

$$\begin{aligned}&\exists t_e \in \mathbb{R}_{\geq 0}\ \forall i \in [\mathcal{N}]\ \exists \mathsf{m} \in \mathsf{L}\ \forall t_p \leq t_e : \\ &\quad \mathbf{flow}(\mathsf{m}, \mathbf{v}.\mathsf{X}[i], t_p) \models \mathbf{inv}(\mathsf{m}, i) \wedge \\ &\quad (\mathbf{flow}(\mathsf{m}, \mathbf{v}.\mathsf{X}[i], t_p) \models \mathbf{stop}(\mathsf{m}, i) \Rightarrow t_p = t_e) \wedge \\ &\quad \mathbf{v}'.\mathsf{X}[i] \in \mathbf{flow}(\mathsf{m}, \mathbf{v}.\mathsf{X}[i], t_e).\end{aligned}$$

For each $i \in [\mathcal{N}]$ and each real variable $x[i] \in \mathsf{X}[i]$, $\mathbf{v}.x[i]$ must evolve to the valuations $\mathbf{v}'.x[i]$, in exactly t_e time in some location $\mathsf{m} \in \mathsf{L}$ according to the flow rates allowed for $x[i]$ in that location. All intermediate states along the trajectory must also satisfy the invariant $\mathbf{inv}(\mathsf{m}, i)$, and if an intermediate state satisfies $\mathbf{stop}(\mathsf{m}, i)$, then that state must be $\mathbf{v}'$ (that is, the end of a trajectory).

[4] We have excluded continuous global variables to make the presentation clearer.

[5] This is an overapproximation of the set of solutions of the rectangular differential inclusion, as it excludes the requirement that the time derivative of any solution is in the differential inclusion $\dot{x}[i] \in [lb, ub]$.

If no flow rate is specified for some variable $x[i] \in \mathsf{X}[i]$, then $x[i]$ is assumed to remain constant along the trajectory (that is, $\dot{x}[i] = 0$). If no invariant is specified, then it is assumed to be *true*, which specifies that the automaton *may* remain indefinitely in the corresponding location. If no stopping condition is specified, it is assumed to be *false*, which will specify allowing real time to elapse indefinitely in the corresponding location.

Together, the components of a trajectory statement define how variables of $\mathcal{A}(i)$ behave over intervals of time. For Fischer, the trajectory statement for try is:

$$\textbf{inv} : x[i] <= A$$

$$\textbf{stop} : x[i] = A$$

$$\textbf{flowrate} : x[i]\_dot >= lb \text{ and } x[i]\_dot <= ub$$

which specifies the same differential inclusion on $x[i]$ as in the location wait, but while $q[i] = \mathsf{try}$. In addition, the invariant requires that the automaton with index i can have $q[i] = \mathsf{try}$ only as long as $x[i] \leq A$. The stopping condition requires that if $x[i] = A$, then real time cannot continue to elapse.[6]

2.4.4 Executions, Invariants, and Inductive Invariants

An execution of the network $\mathcal{A}^{\mathcal{N}}$ models a particular behavior of all the automata in the network. An *execution* of $\mathcal{A}^{\mathcal{N}}$ is a sequence of states $\alpha = \mathbf{v}_0, \mathbf{v}_1, \ldots$ such that $\mathbf{v}_0 \in \Theta^{\mathcal{N}}$, and for each index k appearing in the sequence $(\mathbf{v}_k, \mathbf{v}_{k+1}) \in T^{\mathcal{N}}$. A state $\mathbf{x}$ is *reachable* if there is a finite execution ending with $\mathbf{x}$. The set of reachable states for $\mathcal{A}^{\mathcal{N}}$ is $\mathsf{Reach}(\mathcal{A}^{\mathcal{N}})$. The set of reachable states for $\mathcal{A}^{\mathcal{N}}$ starting from an arbitrary subset $\mathbf{V}_0 \subseteq Q^{\mathcal{N}}$ is $\mathsf{Reach}(\mathcal{A}^{\mathcal{N}}, \mathbf{V}_0)$.

An invariant for $\mathcal{A}^{\mathcal{N}}$ is any set of states that contains $\mathsf{Reach}(\mathcal{A}^{\mathcal{N}})$. In general, any assertion over the variables of the automata in $\mathcal{A}^{\mathcal{N}}$ defines a subset of $Q^{\mathcal{N}}$. The dependence of such assertions on $\mathcal{N}$ is made explicit by using names like $\zeta(\mathcal{N})$. A network $\mathcal{A}^{\mathcal{N}}$ is safe with respect to an assertion $\zeta(\mathcal{N})$ if all its reachable states satisfy it, that is, $\mathsf{Reach}(\mathcal{A}^{\mathcal{N}}) \subseteq [\![\zeta(\mathcal{N})]\!]$.

[6] In this example, the stopping condition is redundant. However, if the flow equations allow time to elapse while the continuous state remains at the boundary of the invariant condition, the stopping condition allows for modeling *urgent transitions*. The stopping condition can force time to stop, which can force transitions to occur.

Given a template hybrid automaton $\mathcal{A}(\mathcal{N}, i)$ and a property $\zeta(\mathcal{N})$, in this dissertation, we develop techniques for proving for all $\mathcal{N} \in \mathbb{N}$, that every network is safe—that is, $\forall \mathcal{N} \in \mathbb{N}$, $\mathsf{Reach}(\mathcal{A}^{\mathcal{N}}) \subseteq [\![\zeta(\mathcal{N})]\!]$. To prove that $\mathcal{A}^{\mathcal{N}}$ is safe with respect to some unsafe set or property—that is, $\neg\zeta(\mathcal{N})$—it suffices to find an invariant $\Gamma(\mathcal{N}) \subseteq Q^{\mathcal{N}}$ such that $[\![\Gamma(\mathcal{N})]\!] \cap [\![\neg\zeta(\mathcal{N})]\!] = \emptyset$.

Several subclasses of hybrid automata have been identified for which safety verification by computing reachable sets is decidable—such as initialized rectangular hybrid automata (IRHA) [30, 111] and order-minimal (o-minimal) hybrid automata [140]—and several automated model checking tools have been developed, such as HyTech [37], PHAVer [39], and SpaceEx [40]. However, the general model checking of even safety properties for hybrid automata is undecidable, so a standard approach is to use overapproximations of reachable states for checking safety properties, such as the methods implemented in PHAVer for affine (linear) dynamics [39] and SpaceEx for affine dynamics as well [40] using the Le Geurnic-Girard (LGG) algorithm [141]. An alternative approach is to prove stronger inductive invariant assertions that imply a desired safety property, as originally used in Floyd-Hoare proofs [142, 143] and the predicate transformers of Dijkstra [144].

Definition 2.4 *An assertion $\Gamma(\mathcal{N})$ is an* inductive invariant *for the parameterized network $\mathcal{A}^{\mathcal{N}}$ if, for all $\mathcal{N} \in \mathbb{N}$, the following conditions hold:*

(A) ***initiation****: for each initial state $\mathbf{v} \in \Theta^{\mathcal{N}} \Rightarrow \mathbf{v} \models \Gamma(\mathcal{N})$,*

(B) ***discrete transition consecution****: for each discrete transition $(\mathbf{v}, \mathbf{v}') \in \mathcal{D}^{\mathcal{N}}$, if $\mathbf{v} \models \Gamma(\mathcal{N})$, then $\mathbf{v}' \models \Gamma(\mathcal{N})$, and*

(C) ***continuous trajectory consecution****: for each trajectory $(\mathbf{v}, \mathbf{v}') \in \mathcal{T}^{\mathcal{N}}$, if $\mathbf{v} \models \Gamma(\mathcal{N})$, then $\mathbf{v}' \models \Gamma(\mathcal{N})$.*

Proving that a parameterized network satisfies a property is the *uniform verification* problem.

Definition 2.5 *The* uniform verification problem *is proving for any $\mathcal{N} \in \mathbb{N}$, for a property $\Gamma(\mathcal{N})$ and parameterized network $\mathcal{A}^{\mathcal{N}}$, that $\mathcal{A}^{\mathcal{N}}$ satisfies $\Gamma(\mathcal{N})$, written $\mathcal{A}^{\mathcal{N}} \models \Gamma(\mathcal{N})$.*

The problem of uniform verification of parameterized networks is over arbitrary compositions of potentially infinite-state automata, so in general we may need to query a theorem prover to check conditions (A), (B), and (C).

The next standard theorem states that if an assertion $\Gamma(\mathcal{N})$ is an inductive invariant—that is, $\Gamma(\mathcal{N})$ satisfies the conditions for inductive invariance (Definition 2.4)—then $\Gamma(\mathcal{N})$ is an invariant [9, 23].

Theorem 2.6 *If* $\forall \mathcal{N} \in \mathbb{N}$, $\Gamma(\mathcal{N})$ *is an inductive invariant, then it is also an invariant:*

$$\forall \mathcal{N} \in \mathbb{N}, \mathsf{Reach}(\mathcal{A}^{\mathcal{N}}) \subseteq [\![\Gamma(\mathcal{N})]\!] .$$

Proof: Fix an arbitrary $\mathcal{N} \in \mathbb{N}$ and consider any reachable state $\mathbf{x} \in \mathsf{Reach}(\mathcal{A}^{\mathcal{N}})$. By definition of reachable state, there exists some execution α with the last state in the execution ending in $\mathbf{x}$, so $\alpha = \mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}$. The proof continues by induction along the sequence states in the execution α.

In the base case, $\mathbf{x}_0$ is an initial state, so $\mathbf{x}_0 \in \Theta^{\mathcal{N}}$, and thus $\mathbf{x}_0 \in [\![\Gamma(\mathcal{N})]\!]$ by the initiation condition (Definition 2.4, (A)). Consider an arbitrary state $\mathbf{x} \in \alpha$, and the induction step is composed of two cases, considering either any discrete transition or any trajectory. If $(\mathbf{x}, \mathbf{x}') \in \mathcal{D}^{\mathcal{N}}$, then by the inductive hypothesis, we have that $\mathbf{x} \in [\![\Gamma(\mathcal{N})]\!]$, and applying the transition consecution condition (Definition 2.4, (B)), we have $\mathbf{x}' \in [\![\Gamma(\mathcal{N})]\!]$. Otherwise, if $(\mathbf{x}, \mathbf{x}') \in \mathcal{T}^{\mathcal{N}}$, then by the inductive hypothesis, we have that $\mathbf{x} \in [\![\Gamma(\mathcal{N})]\!]$, and applying the trajectory consecution condition (Definition 2.4, (C)), we have $\mathbf{x}' \in [\![\Gamma(\mathcal{N})]\!]$. ■

Theorem provers such as PVS have been augmented with support for verifying such networks in [34, 68, 145]. The KeYmaera theorem prover also has support for verifying this type of networks [14, 35]. These environments provide partially automatic means of proving inductive invariants of networks $\mathcal{A}^{\mathcal{N}}$.

It is well known that the converse of Theorem 2.6 does not hold. We next illustrate this with a counterexample using the Fischer example (Figure 2.1).

Counterexample 2.7 *Mutual exclusion is specified as* $\phi(\mathcal{N}) \triangleq \forall i, j \in [\mathcal{N}] : i \neq j \Rightarrow (q[i] \neq$

$\mathsf{cs} \vee q[j] \neq \mathsf{cs}$). *Suppose that* $\phi(\mathcal{N})$ *is an invariant of* Fischer,[7] *and we will show that it is not an inductive invariant.*

We construct a state $\mathbf{x} \in [\![\phi(\mathcal{N})]\!]$ *as follows. Let* i *be the index of an automaton with local variable valuations satisfying* $\mathbf{x}.q[i] = \mathsf{cs}$. *Let* j *be the index of an automaton with local variable valuations satisfying* $\mathbf{x}.q[j] \neq \mathsf{cs}$, *specifically suppose* $\mathbf{x}.q[j] = \mathsf{wait}$. *Consider the transition* t *from* wait *to* cs, *and we have* $\mathbf{x} \models \mathbf{grd}(\mathsf{t}, j)$, *so the transition* t *is enabled and may be taken by automaton* j. *The effect* $\mathbf{eff}(\mathsf{t}, j)$ *sets* $\mathbf{x}'.q[j] = \mathsf{cs}$ *while leaving* $\mathbf{x}'.q[i] = \mathbf{x}.q[i] = \mathsf{cs}$, *so there are two automata in the critical section which violates the invariant* $\phi(\mathcal{N})$. *Therefore,* $\phi(\mathcal{N})$ *is not an inductive invariant since transition consecution is violated (Definition 2.4, (B)).*

If we can find an assertion $\Gamma(\mathcal{N})$ that is an inductive invariant and implies some desired safety property $\zeta(\mathcal{N})$, we say that $\Gamma(\mathcal{N})$ is sufficient to prove the safety property $\zeta(\mathcal{N})$.

Definition 2.8 *For any* $\mathcal{N} \in \mathbb{N}$, *an assertion* $\Gamma(\mathcal{N})$ *is sufficient to prove a safety property* $\zeta(\mathcal{N})$ *if* $\Gamma(\mathcal{N})$ *is an inductive invariant and* $\Gamma(\mathcal{N}) \Rightarrow \zeta(\mathcal{N})$, *so that* $[\![\zeta(\mathcal{N})]\!] \subseteq [\![\Gamma(\mathcal{N})]\!]$.

For Fischer, the following inductive invariant is sufficient to prove that mutual exclusion is an invariant.

$$\forall i \in [\mathcal{N}] : q[i] = \mathsf{try} \Rightarrow x[i] \leq A \wedge \tag{2.4}$$

$$\forall i, j \in [\mathcal{N}] : (q[i] = \mathsf{wait} \wedge g = i \wedge q[j] = \mathsf{try}) \Rightarrow (B - A) > (x[i] - x[j]) \wedge \tag{2.5}$$

$$\forall i, j \in [\mathcal{N}] : q[i] = \mathsf{cs} \Rightarrow (g = i \wedge q[j] \neq \mathsf{try}). \tag{2.6}$$

When trying to prove a safety property by coming up with an inductive invariant, one usually reasons in reverse as follows. Why is it that mutual exclusion is not inductive for Fischer? As described in Counterexample 2.7, it is because, when assuming only knowledge that mutual exclusion is satisfied, the guard of the transition $\mathsf{t}(\mathsf{wait}, \mathsf{cs})$ can be enabled, since mutual exclusion does not explicitly specify that $g \neq j$. What information—constraints on states—is required to prevent the transition $\mathsf{t}(\mathsf{wait}, \mathsf{cs})$ from being able to occur? By

[7] Mutual exclusion is in fact an invariant of Fischer, but we assume it is an invariant here simply to show that an invariant is not necessarily inductive.

considering each transition in Trans_i, we see that $\chi = \forall i, j \in [\mathcal{N}] : (q[i] = \mathsf{cs}) \Rightarrow (g = i \wedge q[j] \neq \mathsf{try})$ is sufficient to prove mutual exclusion. However, this process repeats, since χ itself is not an inductive invariant. By repeating this process of reasoning backward, one may[8] be able to come up with the conditions of Equation 2.6 that are sufficient to prove that the mutual exclusion safety property is invariant.

This manual refinement process may be useful, but it does require human intervention, and is not guaranteed to produce useful invariants. We introduce methods for automatically finding inductive invariants in Chapter 6.

2.5 Summary

In this chapter, we describe a modeling framework for analyzing parameterized networks of hybrid automata. We informally introduce the class of systems in Section 2.2, the syntactic structure of a template hybrid automaton $\mathcal{A}(\mathcal{N}, i)$ was specified in Section 2.3. The formal semantics of parameterized networks composed of copies of the template are defined next in Section 2.4, along with definitions of safety properties and an overview of the inductive invariance proofs for establishing safety properties. Using the template $\mathcal{A}(\mathcal{N}, i)$ as input, *Passel* represents the semantics of the hybrid automaton network $\mathcal{A}^{\mathcal{N}}$ composed of arbitrarily many copies of $\mathcal{A}(\mathcal{N}, i)$, although *Passel* never explicitly computes this composition, and only encodes the formulas describing the semantics of the discrete transitions and continuous trajectories (see Chapter 7 for more details).

[8] This is not guaranteed to find an inductive invariant sufficient to prove a property.

Chapter 3

Parameterized Reachability Analysis: A Case Study on Distributed Air Traffic Control

In this chapter, we present the formal modeling and automatic parameterized verification of a distributed air traffic control protocol called the Small Aircraft Transportation System (SATS). A simplified version of SATS was presented earlier in Chapter 2, Section 2.3.8. The verification methodology relies on computing the set of backward reachable states from the set of unsafe states to a fixed-point, and checking emptiness of the intersection of these reachable states and the initial set of states. We use the Model Checker Modulo Theories (MCMT) tool [54] to implement this methodology for the case study.[1]

3.1 Introduction

This chapter presents a distributed cyber-physical system (DCPS), namely a distributed air traffic control protocol, for which we automatically verify several non-trivial properties for arbitrarily many participating aircraft. The Small Aircraft Transportation System (SATS) was developed with the goal of increasing access to small airports that potentially do not have control towers nor radar. Instead, the aircraft rely on (a) receiving landing sequence information from an automated airport management module (AMM) located at the airport, and (b) communicating with one another to determine landing orders and perform landings. The overall operation must satisfy a variety of safety properties—such as, between each aircraft, there is always a sufficiently large physical separation.

The model presented in this chapter differs from the one presented in Section 2.3.8 (Figure 2.3). In particular, the model we present next uses counters instead of pointers and has timed dynamics instead of rectangular differential inclusions. These choices were made to

[1] This chapter is based in part on prior work [2], portions of which are reprinted here with permission.

encode the model in the Model Checker Modulo Theories (MCMT) tool [32, 54] that implements the backward reachability method we describe in this chapter. We discuss related work on verifying other DCPS like air traffic control systems and automotive systems and give an overview of the relevant literature on uniform verification in Section 1.3.

3.1.1 SATS Overview

In SATS, each aircraft i has a one-dimensional position evolving with time, representing its distance from approach to the runway. Consider an automaton template $\mathcal{A}(\mathcal{N}, i)$ for some $i \in [\mathcal{N}]$. Recall that we denote $\mathcal{A}(\mathcal{N}, i)$ as $\mathcal{A}(i)$ when $\mathcal{N}$ is clear from context. There is a single runway where the aircraft need to land. Refer to Figure 3.1 for an overhead view of the landing area. There are two entry points to the runway, coming from the left or right of it. Each aircraft begins flying and may enter either the left or right cyclic holding patterns called *holding zones*. In this step, an aircraft is assigned a *sequence number*, which is the order in which the aircraft should land. While in the holding pattern, the aircraft are assumed to have a safe separation, and hence the values of the continuous positions do not matter. However, an aircraft may attempt an approach to the runway, at which point it exits the holding zone, begins on a path toward the runway, and the values of the continuous positions become significant. Upon attempting to land, the aircraft may either land on the runway and subsequently taxi away, or it may *miss the approach* and return to a cyclic holding zone.

Communication and Sensing Requirements. Next we present the aircraft and airport communication and sensing requirements that will lead into our model of the operation of SATS. The Airport Management Module (AMM) is a ground-based automation system that would typically be located at an airport and provides sequencing information to pilots over a ground-to-air datalink [25, 26]. The AMM is the main centralized communication component of SATS, and all other communication is decentralized and done either (a) between pilots over voice radio, or (b) between aircraft control software via air-to-air datalinks.

Each aircraft in SATS is required to have the following sensing and communication capabilities [138]: (a) global positioning system (GPS) receiver, (b) air-to-ground datalink communication, for broadcast and receipt of AMM messages, (c) air-to-air datalink commu-

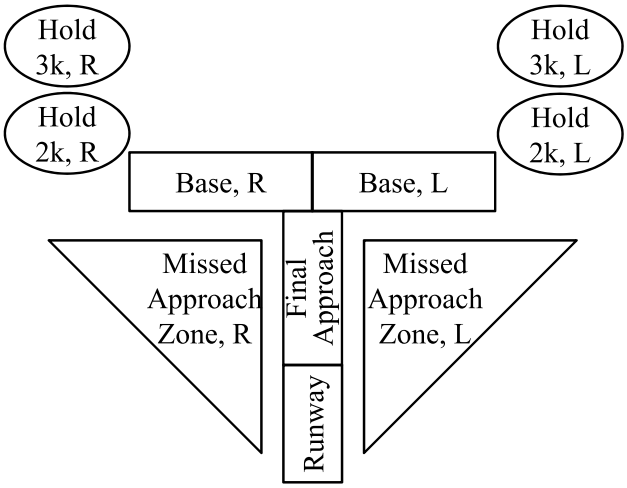

Figure 3.1: SATS viewed from above, except the holding zones at different elevations would be directly atop one another. There are two sides to approach the runway from, left (L) and right (R), and right and left are reversed in the image for the pilot's orientation. The safe spacing property matters when on the base, final, or missed zones, but not in other zones.

nication, (d) cockpit Display of Traffic Information, which provides a pilot the location of his/her aircraft and other nearby aircraft, (e) software to conduct the landing procedures, which informs a pilot who to follow and where to go by displaying sequencing information from AMM and uses Conflict Detection and Alerting Algorithms, and (f) voice communication radio.

Several desired properties are defined for SATS informally in the technical reports describing the system [25, 26]. The main safety property is *separation assurance*, that is, no two aircraft come closer than a pre-specified distance from one another, and hence never collide. There are also restrictions on the number of aircraft that may simultaneously be approaching the runway.

Outline. In Section 3.2, we present a formal model of SATS as a template hybrid automaton $\mathcal{A}(\mathcal{N}, i)$. In Section 3.3, we introduce a general backward reachability method that we use to verify safety properties for the network $\mathcal{A}^{\mathcal{N}}$ for any $\mathcal{N} \in \mathbb{N}$. In Section 3.4, we describe a simple example to illustrate the reachability method and the conditions under which it terminates. In Section 3.5, we define and verify several safety properties for SATS using the Model Checker Modulo Theories (MCMT) [54] tool that implements this backward reachability method, and we conclude in Section 3.6.

3.2 Formal Model of the Small Aircraft Transportation System

In this section, we present a formal model of SATS.

We describe the protocol followed by the i^{th} aircraft, which is modeled as hybrid automaton template $\mathcal{A}(\mathcal{N}, i)$ (recall Definition 2.1). Refer to Figure 3.2 for states of SATS and refer to Section 3.2.2 for detailed definitions of the transitions. For aircraft i, $q[i]$ is its current flight mode. Aircraft i starts in the flying mode ($q[i] = \mathbf{fly}$). It decides to land *nondeterministically* by entering the left or right holding zone at 3000 feet ($q[i] \in \{\mathbf{h3^r}, \mathbf{h3^l}\}$). Upon entry, i is assigned a sequence number (0 if there are no other aircraft in the system, or $c + 1$, where c is the value of the sequence number assigned to the last aircraft attempting to land). Subsequently, aircraft i *may* descend to the holding zone at 2000 feet ($q[i] \in \{\mathbf{h2^r}, \mathbf{h2^l}\}$). If there is enough spacing between i and the aircraft with sequence number one less than i's (if one exists), then i transitions to either the base left or right zone ($q[i] \in \{\mathbf{b^r}, \mathbf{b^l}\}$). Aircraft i is never forced to transition from a holding pattern to an approach toward the runway. Rather, any aircraft *may* nondeterministically begin the approach, so long as the spacing condition is satisfied.

After traveling down the base zone for enough distance ($x[i] \geq L_B$, where L_B is the length of the base zone), i moves to the final approach zone ($q[i] = \mathbf{fin}$). Finally, after traversing the length of the final zone ($x[i] \geq L_F$, where L_F is the length of the final zone), an aircraft *may* either: (a) land and go to the runway ($q[i] = \mathbf{run}$), or (b) miss its approach ($q[i] \in \{\mathbf{m^r}, \mathbf{m^l}\}$). Then, after traversing the length of the missed zone ($x[i] \geq L_M$, where L_M is the length of the missed zone), i restarts the process of moving from holding to final. If aircraft i misses its attempt to land, it is assigned a new sequence number at the end. Allowing aircraft to miss an approach is one reason that several of the properties to be introduced below are non-trivial to verify. The missed approach is initiated by the pilot if for any reason a safe landing cannot be assured (e.g., due to unforeseen weather changes, flying too fast to stop on the runway length, unknown obstacles on the runway, etc.). The only locations where the continuous position $x[i]$ matters are the base, final, and missed zones.

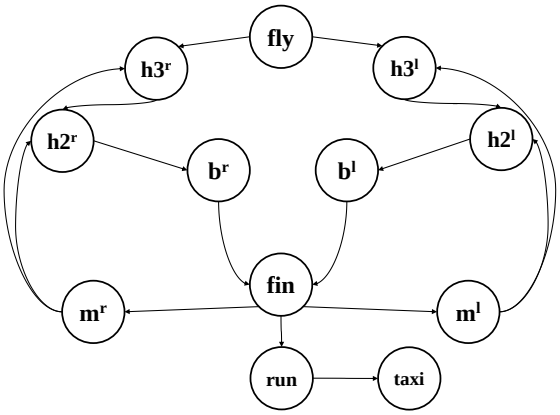

Figure 3.2: SATS locations, transitions, and invariants for aircraft i. The continuous variable $x[i]$ only matters in states $\mathbf{b^r}$, $\mathbf{b^l}$, **fin**, $\mathbf{m^r}$, and $\mathbf{m^l}$, which correspond to when i is attempting to land on the runway by reaching location **run**. Invariants for continuous variables in locations are captured by, for instance, $x[i] \leq L_B$, etc., for some $L_B > 0$.

3.2.1 Hybrid Automaton Template of Aircraft i in SATS

We specify a single aircraft in SATS as a hybrid automaton template $\mathcal{A}(\mathcal{N}, i)$, which is used to define the semantics of the network $\mathcal{A}^{\mathcal{N}}$ (recall Section 2.4). The specification is a translation of the PVS model used in [27], with the following two exceptions. First, we do not model lateral entry zones, where an aircraft could go from a lateral entry zone state to the holding zone at either 3000 or 2000 feet, instead of having to start from the vertical entry to the 3000 foot holding zone. Second, we use timed dynamics ($\dot{x} = 1$) instead of rectangular dynamics ($\dot{x} \in [a, b]$ for $a \leq b$). Modeling and verifying lateral entry zones would be relatively easy, but doing so for rectangular dynamics would require significant additional work.

There are four local variables (lines 7 through 10) and one global variable (line 11). Four of these are discrete variables $q[i]$, $m[i]$, $s[i]$, and c, and one is a continuous variable $x[i]$. The control *location* $q[i]$ has type:

$$\mathsf{L} \triangleq \{\mathbf{fly}, \mathbf{h3^r}, \mathbf{h2^r}, \mathbf{h3^l}, \mathbf{h2^l}, \mathbf{b^r}, \mathbf{b^l}, \mathbf{fin}, \mathbf{m^r}, \mathbf{m^l}, \mathbf{run}, \mathbf{taxi}\}.$$

The elements of L represent:

(a) **fly**: the aircraft is flying,

(b) $\mathbf{h3^r}, \mathbf{h2^r}, \mathbf{h3^l}, \mathbf{h2^l}$: the aircraft is in the holding zone on the right side at 3000 feet (2000 feet right side for $\mathbf{h2^r}$, and left for $\mathbf{h3^l}$ and $\mathbf{h2^l}$),

(c) $\mathbf{b^r}, \mathbf{b^l}$: attempting to land in the base segment on the right (left) side,

(d) **fin**: on the final approach to the runway,

(e) $\mathbf{m^r}, \mathbf{m^l}$: missed or aborted the landing attempt and on the right (left) side,

(f) **run**: landed on the runway, and

(g) **taxi**: taxied off the runway to a gate.

The next variable is the missed approach $m[i]$ of type $Side \triangleq \{left, right\}$ that indicates which side an aircraft will go to if it aborts its landing attempt.[2] We abbreviate *left* as l and *right* as r when clear. The third variable is the sequence number $s[i]$ of type $\mathbb{N}$, which represents the sequence in which aircraft should land (where $s[i] = 1$ would mean i should land first, and so on). The last discrete variable is a global variable c of type $\mathbb{N}$, which is a counter tracking the last sequence number of an aircraft entering the system.[3]

There is a single continuous variable $x[i]$ of type $\mathbb{R}$ that represents the one-dimensional distance aircraft i has traveled from the physical starting point of a zone. In Figure 3.1, this is the distance measured along one dimension from the start of, to the end of, each of the base region, final approach region, and missed approach zones.

The initial assertion is:

$$\mathsf{Init}_i \triangleq \forall i \in \mathcal{N} : (q[i] = \mathbf{fly} \wedge x[i] = 0 \wedge s[i] = 0 \wedge c = 0).$$

The initial value of $m[i]$ is irrelevant, as it is updated before use. There are discrete transitions between locations for SATS as shown in Figure 3.2.

[2]This is modeled as a Boolean variable where $left = 0$ and $right = 1$, each of which are written in short as l and r, respectively.

[3]We assume that all $\mathcal{N}$ automata have the same valuation of c, so we do not index it with a subscript. This fits within the timed network framework of [32, 47].

```
1  parameter name='L_B' type='real'  // base zone length
   parameter name='L_F' type='real'  // final zone length
3  parameter name='L_M' type='real'  // missed zone length
   parameter name='L_S' type='real'  // separation spacing
5
   automaton name='SATS'
7    variable name='q[i]' type='L'        // location local variable
     variable name='s[i]' type='integer' // sequence number
9    variable name='m[i]' type='boolean' // missed approach zone
     variable name='x[i]' type='real'    // continuous local variable
11   variable name='c' type='integer'    // global counter variable

13   location name='fly'
     location name='h3^r'
15   location name='h3^l'
     location name='h2^r'
17   location name='h2^l'
     location name='b^r'
19     inv: x[i] <= L_B
       flowrate: x[i]_dot = 1.0
21   location name='b^l'
       inv: x[i] <= L_B
23     flowrate: x[i]_dot = 1.0
     location name='fin'
25     inv: x[i] <= L_F
       flowrate: x[i]_dot = 1.0
27   location name='m^r'
       inv: x[i] <= L_M
29     flowrate: x[i]_dot = 1.0
     location name='m^l'
31     inv: x[i] <= L_M
       flowrate: x[i]_dot = 1.0
33   location name='run'
     location name='taxi'
35
   initially: forall i (q[i] = fly ∧ x[i] = 0 ∧ s[i] = 0 ∧ c = 0)
```

Figure 3.3: Portion of *Passel* input file with the variables, locations, and initial assertions of the hybrid automaton template $\mathcal{A}(i)$ specifying SATS.

3.2.2 SATS Transitions and Trajectories

We now go through each of the transitions in the operation of SATS.

All aircraft begin flying, and may enter the system by transitioning from **fly** to the left or right holding zone at 3000 feet. For aircraft i, for a transition t from **fly** to $\mathbf{h3^r}$ (and symmetrically for $\mathbf{h3^l}$) is specified on line 1. Note that we simplify notation and do not write identity resets for any variable not appearing in a reset, although these must appear in the formulas defining the semantics (recall Section 2.4.3), as must be specified in the input for MCMT. For example, we dropped $m[j]' = m[j]$ and $x[j]' = x[j]$ from $\mathbf{ugrd}(\mathsf{t})$ and $x[i]' = x[i]$ from $\mathbf{eff}(\mathsf{t})$. We recall that the semantics from Section 2.4.3 of the transitions are encoded as first-order logic formula using quantifiers as: $\exists i \in [\mathcal{N}] : \mathbf{grd}(\mathsf{t}) \wedge \mathbf{eff}(\mathsf{t}) \wedge \forall j \in [\mathcal{N}] : \mathbf{ugrd}(\mathsf{t})$.

```
    transition from='fly' to='h3^r'
      eff: s[i]' = c + 1 ∧ c' = c + 1
      ugrd: q[j] ≠ h3^r

    transition from='h3^r' to='h2^r'
      ugrd: q[j] ≠ h2^r

    transition from='h2^r' to='b^r'
      eff: x[i]' = 0
      ugrd: s[i] ≠ 1 ⇒ s[j] = s[i] − 1 ⇒ x[j] − x[i] ≥ L_S

    transition from='b^r' to='fin'
      eff: x[i]' = 0
      grd: x[i] ≥ L_B

    transition from='fin' to='m^r'
      eff: x[i]' = 0
      grd: x[i] ≥ L_F ∧ m[i] = r

    transition from='fin' to='run'
      eff: x[i]' = 0
      grd: x[i] ≥ L_F ∧ m[i] = r
      ugrd: q[j] ≠ run

    transition from='run' to='taxi'

    transition from='m^r' to='h3^r'
      eff: x[i]' = 0 ∧ s[i]' = c
      grd: x[i] ≥ L_M
      ugrd: q[j] ≠ h3^r ∧ (j ≠ i ⇒ s[j]' = s[j] − 1)

    transition from='m^r' to='h2^r'
      eff: x[i]' = 0 ∧ s[i]' = c
      grd: x[i] ≥ L_M
      ugrd: q[j] ≠ h3^r ∧ q[j] ≠ h2^r ∧ (j ≠ i ⇒ s[j]' = s[j] − 1)
```

Figure 3.4: Portion of *Passel* input file with the transitions of the hybrid automaton template $\mathcal{A}(i)$ specifying SATS. Only the transitions for the right side are shown, and the left side transitions are specified analogously by substituting r for l.

This is equivalent to the following formula used by MCMT:

$$\begin{aligned}\tau_{\mathbf{fly}\to\mathbf{h3^l}} \triangleq\ & \forall j \in [\mathcal{N}] : (q[j] \neq \mathbf{h3^r}) \wedge \exists i \in [\mathcal{N}] : (q[i] = \mathbf{fly} \wedge \\ & q' = \lambda j.(\text{if } j = i \text{ then } \mathbf{h3^r} \text{ else } q[j]) \wedge \\ & s' = \lambda j.(\text{if } j = i \text{ then } c + 1 \text{ else } s[j]) \wedge c' = c + 1),\end{aligned}$$

where the λj notation is equivalent to $\forall j$ and is used to ensure every component of (the vectors) q and s are defined.

We now describe the remaining transitions without going through this syntactic translation. Once in a (left or right) holding zone at 3000 feet, an aircraft may descend to the holding zone on the same side at 2000 feet. For the right side (and symmetrically for the

left), this is described as as a transition t from $\mathbf{h3^r}$ to $\mathbf{h2^r}$ (and symmetrically for $\mathbf{h3^r}$ to $\mathbf{h2^r}$) on line 5.

Once in a (left or right) holding zone at 2000 feet, an aircraft may begin the approach to the runway by transitioning to the base zone on the same side. For the right side (and symmetrically for the left), this is defined as the transition t from $\mathbf{h2^r}$ to $\mathbf{b^r}$ (and symmetrically, $\mathbf{h2^l}$ to $\mathbf{b^l}$) on line 8.

Once an aircraft on the left or right base and on approach to the runway has traveled the length L_B of the base zone, it must enter the final approach zone. This must requirement makes this an urgent transition. This is specified for the right side (and symmetrically, left) $\mathbf{b^r}$ to **fin** by the transition on line 12.

Once an aircraft on final approach travels the length L_F of the final approach zone, it must either miss the approach and enter the missed approach zone on the appropriate side, or it may land on the runway. Note that the side the aircraft misses to is defined by the value of the variable $m[i]$, and is not necessarily the same side on which the aircraft initiated the approach to the final zone. This is specified by the transition from the final approach **fin** to the right missed zone $\mathbf{m^r}$ (and symmetrically, left missed zone $\mathbf{m^l}$) on line 16.

After traveling the length L_M of the missed zone, an aircraft must transition to the lowest altitude holding zone without any aircraft in it on the same side as the missed zone. The right missed approach zone (and symmetrically, left) to the holding zone is specified with two transitions. First, is the transition t from $\mathbf{m^r}$ to $\mathbf{h3^r}$ on line 27. Second, is the transition t from $\mathbf{m^r}$ to $\mathbf{h2^r}$ on line 32. For the miss transitions, we model the update values of $m[i]$ nondeterministically, which departs from the actual SATS specification, but is an abstraction. Also, observe that we use the universal guard to decrease the value of the sequence numbers of the other aircraft.

Alternatively, if aircraft i does not miss, then the transition t from the final approach **fin** to the runway **run** is specified on line 20. An aircraft on the runway may then taxi away, defined as the transition t from **run** to **taxi** on line 25.

Since this model of SATS utilizes timed dynamics, trajectories are defined according to

the following first-order logic formula,

$$\begin{aligned}\exists t_e \in \mathbb{R}_{\geq 0}\ \forall j \in [\mathcal{N}] : x[j]' = x[j] + t_e \wedge \\ (q[j] \in \{\mathbf{b^r}, \mathbf{b^l}\} \Rightarrow x[j]' \leq L_B) \wedge \\ (q[j] = \mathbf{fin} \Rightarrow x[j]' \leq L_F) \wedge \\ (q[j] \in \{\mathbf{m^r}, \mathbf{m^l}\} \Rightarrow x[j]' \leq L_M).\end{aligned}$$

This formula models that time elapses in the same amount t_e for every aircraft, and thus their continuous positions evolve according to trajectories of the same length. Recall the semantics of trajectories defined in Section 2.4.3, and observe that this models many trajectories.

3.3 Verification by Backward Reachability

Given a set $[\![\upsilon(\mathcal{N})]\!]$ of unsafe states, an automaton network $\mathcal{A}^{\mathcal{N}}$ is said to be safe with respect to $\upsilon(\mathcal{N})$ if the set is never reached by $\mathcal{A}^{\mathcal{N}}$, that is, $[\![\upsilon(\mathcal{N})]\!] \cap \mathsf{Reach}(\mathcal{A}^{\mathcal{N}}) = \emptyset$. Equivalently, the safety property $\zeta(\mathcal{N}) = \neg\upsilon(\mathcal{N})$ is satisfied by $\mathcal{A}^{\mathcal{N}}$ if $\neg\upsilon(\mathcal{N})$ contains the reachable states $\mathsf{Reach}(\mathcal{A}^{\mathcal{N}})$ (recall Section 2.4.4). To establish that an automaton network $\mathcal{A}^{\mathcal{N}}$ is safe with respect to a property $\zeta(\mathcal{N})$, we can show that the set $[\![\zeta(\mathcal{N})]\!]$ is invariant, that is, $\zeta(\mathcal{N})$ contains all the reachable states $\mathsf{Reach}(\mathcal{A}^{\mathcal{N}})$.

For SATS, one safety property is separation assurance—that is, no two aircraft ever come too close together—which can be written as:

$$\zeta(\mathcal{N}) \triangleq \forall i, j \in [\mathcal{N}] : i \neq j \wedge s[i] = s[j] - 1 \Rightarrow x[i] - x[j] \geq L_S,$$

where $s[i] = s[j] - 1$ indicates that aircraft i is ahead of aircraft j in the landing sequence, and $L_S > 0$ is the minimum separation desired between aircraft i and j. We can can define the unsafe property as the negation $\neg\zeta(\mathcal{N})$, which for separation assurance would assert that there are two aircraft too close together. In general, to prove a safety property $\zeta(\mathcal{N})$ automatically, it suffices to take the negation of the safety property, and check that the set of backward reachable states from $[\![\neg\zeta(\mathcal{N})]\!]$ has an empty intersection with the initial set of

```
1       k := 0
        φ_k := υ(N), where υ(N) ≡ ¬ζ(N)
3       ρ_k := φ_k
        σ_k := ∅
5
        while true {
7         if ρ_k ∧ Init_i satisfiable // safety check
            return unsafe and σ_k counterexample
9         k := k + 1
          for each t ∈ Trans_i {
11          φ_k := φ_k ∨ Pre_t(φ_{k−1}) ≡ φ_k ∨ ∃V'_i.(t(V_i, V'_i) ∧ φ_{k−1}(V'_i))
            σ_k(t) := σ_k(t) ∪ t  // keep tree of valid executions
13        }
          ρ_k := ρ_{k−1} ∨ φ_k
15
          if ¬ (ρ_k ⇒ ρ_{k−1}) unsatisfiable // fixed−point check
17          return safe
        }
```

Figure 3.5: Backward reachability algorithm used by MCMT.

states, $[\![\mathsf{Init}_i]\!]$. This is the method used by the verification tool we use, the Model Checker Modulo Theories (MCMT) [32,33,53,54]. If the intersection of the backward reachable states and the initial states is empty and the backward reachability process terminates—that is, the backward reachability computation reaches a fixed-point and no new states are added on a preimage computation—then $\mathcal{A}^{\mathcal{N}}$ satisfies $\zeta(\mathcal{N})$ for any $\mathcal{N} \in \mathbb{N}$.

Under the assumption that the desired safety property is an index quantified *Passel* assertion, the preimage computation from the set of unsafe states is not much different than that for a non-parameterized system. For instance, consider the negation of the separation assurance property, which states there are two aircraft less than the safety distance apart,

$$\neg\zeta(\mathcal{N}) \triangleq \exists i, j \in [\mathcal{N}] : i \neq j \wedge s[i] = s[j] - 1 \wedge x[i] - x[j] < L_S.$$

Observe that $\neg\zeta(\mathcal{N})$ is defined in terms of two aircraft being in a particular state. The preimage computation will return a formula with the same existential quantifiers.[4] For instance, the preimage of the formula $\exists i \in [\mathcal{N}] : q[i] = \mathbf{run}$ is roughly—we omit some details to present the intuition—the formula $\exists i \in [\mathcal{N}] : q[i] = \mathbf{fin}$, since the only way for an aircraft to reach the runway is from the final approach zone **fin**. Observe that this does not increase the number of quantified index variables appearing in the formula—that is, the preimage of $\exists i \in [\mathcal{N}] : Q(i)$ is not of the form $\exists i, j \in [\mathcal{N}] : Q(i, j)$.

[4] In general this is not true, but see [135] for when it is.

The main complication is ensuring that the sequence of preimage computations terminates. A detailed theory of when this preimage computation will terminate has been developed for parameterized systems [135,146], and parameterized timed systems [47,48]. Our formulation of SATS is undecidable—that is, a fixed-point may not be reached—for two main reasons. First, we model urgent trajectories, that is, we prevent trajectories from continuing once some condition becomes true [47]. Second, we use universally quantified index variables in some transition guards [51].

Now, why should we expect this process to ever reach a fixed-point? For instance, why is it not possible for new aircraft to continually enter the system? Observe that the unsafe property is of the form $\exists i \in [\mathcal{N}] : \varphi(i)$. With this form of property, all that matters is whether there is some aircraft in the system satisfying $\varphi(i)$. Again, observe that the preimage computation will return a formula of the form $\exists i \in [\mathcal{N}] : Q(i)$. It is essential for termination that the preimage computation does not always add existentially quantified index variables to the formula. If the preimage is $\exists i \in [\mathcal{N}] : Q(i)$, where $Q(i)$ corresponds to a formula not implied by $\varphi(i)$ (or any of the formulas corresponding to already reached states), then we cannot terminate, but otherwise we can. Likewise, if the unsafe property is of the form $\exists i, j \in [\mathcal{N}] : \varphi(i, j)$, then all that matters is whether there are two processes satisfying the formula, etc.

The semi-algorithm takes four inputs: an initial formula Init_i, a formula representing the unsafe set of states $\upsilon(\mathcal{N})$, the transition rules for one automaton template $\mathcal{A}(i)$, and some auxiliary axioms that hold for the parameterized system $\mathcal{A}^{\mathcal{N}}$ (which are useful for asserting data type constraints and already proved safety properties). These inputs are essentially specified as formulas in a restricted subclass of first-order logic. For example, a safety property of SATS is that there is never more than a single aircraft in the left holding zone at 3000 feet, $\mathbf{h3^l}$, and hence the unsafe states are those where there are two or more aircraft in $\mathbf{h3^l}$. More formally, the safety property is $\forall i, j \in [\mathcal{N}] : i \neq j \Rightarrow (q[i] \neq \mathbf{h3^l} \vee q[j] \neq \mathbf{h3^l})$, and the unsafe property is $\exists i, j \in [\mathcal{N}] : i \neq j \wedge q[i] = \mathbf{h3^l} \wedge q[j] = \mathbf{h3^l}$.

Next, we describe how this procedure is implemented by MCMT in the pseudocode shown in Figure 3.5. The parameterized network $\mathcal{A}^{\mathcal{N}}$ evolves according to the set of transitions Trans_i. The algorithm implemented in MCMT processes first-order logic formulas that de-

scribe sets of states. For backwards reachability, the first formula describes a bad (that is, unsafe or illegal) set of states, denoted by $\upsilon(\mathcal{N})$.

For $\mathcal{N} \geq 2$, the network $\mathcal{A}^{\mathcal{N}}$ will be safe if the algorithm reaches a fixed-point and the constraints describing the set of states that may reach $[\![\upsilon(\mathcal{N})]\!]$ do not intersect with the initial set of states, described by $[\![\mathsf{Init}_i]\!]$. Recall $\Theta^{\mathcal{N}} \triangleq [\![\mathsf{Init}_i]\!]$ for any $\mathcal{N} \geq 2$. Let ρ_k be the sequence of formulas starting from $\upsilon(\mathcal{N})$. Defined inductively, $\rho_0 \triangleq \upsilon(\mathcal{N})$ and $\rho_k \triangleq \rho_{k-1} \vee Pre(\phi_{k-1})$. Pre is the preimage of a formula, defined as $Pre_{\mathsf{t}}(\phi_{k-1}) \equiv \exists \mathsf{V}_i'.\, (\mathsf{t}(\mathsf{V}_i, \mathsf{V}_i') \wedge \phi_{k-1}(\mathsf{V}_i'))$ for each transition $\mathsf{t} \in \mathsf{Trans}_i$. Here, $(\mathsf{V}_i, \mathsf{V}')$ highlights that t is over the variables V_i and their primed versions, V_i'. The notation $\phi_{k-1}(\mathsf{V}_i')$ means all variables from V_i appearing in ϕ_{k-1} have been replaced with their primed versions. Thus, ρ_k represents the set of states that can reach the bad set of states in k iterations of the algorithm. Sets of formulas are denoted by ϕ_k for the k^{th} iteration of the backwards reachability.

It is desired that a fixed-point is eventually reached, that is, so that $\rho_k \equiv \rho_{k-1}$. The problem is in general undecidable [28, 29, 47], so it may be the case that no fixed-point is reached. To check if a fixed-point has been reached, one checks if $[\![\rho_k]\!] \subseteq [\![\rho_{k-1}]\!]$. This is equivalent to checking satisfiability of $\neg(\rho_k \Rightarrow \rho_{k-1})$. Conditions for decidability of the safety and fixed-point checks are given in [135], as are conditions for when a fixed-point is guaranteed to exist.

Observe that it is sufficient to consider transitions that we know occur, that is, the ones for which we know the guards are satisfied, as otherwise it will be trivially unsatisfiable, so we exclude these. In an implementation, a tree would be constructed of the disjunctions of the t transitions and those which are unsatisfiable would be pruned from the search path [135]. Next, we will run the reachability algorithm on a simple example by hand to elucidate termination.

3.4 Example: Finite State Automaton with Unreachable Illegal States

We take a brief diversion to illustrate the algorithm used in MCMT with a much simpler example of a parameterized network of finite state automata (FSA) that can be worked out

```
automaton name='FSA'
  variable name='q[i]' type='L'        // location local variable

  location name='b0'
  location name='b1'
  location name='b2'

  transition from='b1' to='b0'

  transition from='b1' to='b1'

  transition from='b2' to='b2'

initially: forall i (q[i] = b2)

property: forall i (q[i] ≠ b0)
```

Figure 3.6: *Passel* input file with the variables, locations, and initial assertions of the automaton template $\mathcal{A}(i)$ specifying the FSA example.

Table 3.1: Safety and fixed-point checks for FSA example.

k	ϕ_k	ρ_k	$\rho_k \wedge \mathsf{Init}_i$	$\neg(\rho_k \Rightarrow \rho_{k-1})$
0	$\exists i : q_i = b_0$	$\exists i : q_i = b_0$	$\exists i : q_i = b_0 \wedge$ $\forall i : q_i = b_2$	undefined
1	$Pre(\exists i : q_i = b_0) \equiv \exists q' \mathsf{t}(q, q') \wedge \phi_0(q')$ $\equiv \exists q' \left(\exists i_1 : q_{i_1} = b_1 \wedge q' = \lambda j.\right.$ $\left.(\text{if } j = i_1 \Rightarrow b_0 \text{ else } q_j) \wedge \exists i_2 : q'_{i_2} = b_0\right)$ $\equiv \exists i : q_i = b_1$	$\exists i : q_i = b_0 \vee \exists i : q_i = b_1$	$(\exists i : q_i = b_0 \vee$ $q_i = b_1)$ $\wedge \forall i : q_i = b_2$ $\equiv$unsatisfiable	$(\exists i : q_i = b_0 \wedge \forall i : q_i \neq b_0)$ $\vee (\exists i : q_i = b_1 \wedge \forall i : q_i \neq b_0)$ $\equiv$satisfiable
2	$Pre(\exists i : q_i = b_1) \equiv \exists q' \mathsf{t}(q, q') \wedge \phi_1(q')$ $\equiv \exists q' \left(\exists i_1 : q_{i_1} = b_1 \wedge q' = \lambda j.\right.$ $\left.(\text{if } j = i_1 \Rightarrow b_1 \text{ else } q_j) \wedge \exists i_2 : q'_{i_2} = b_1\right)$ $\equiv \exists i : q_i = b_1$	$\exists i : q_i = b_0 \vee \exists i : q_i = b_1$ $\vee \exists i : q_i = b_1$ $\equiv \exists i : q_i = b_0 \vee \exists i : q_i = b_1$	$(\exists i : q_i = b_0 \vee$ $q_i = b_1)$ $\wedge \forall i : q_i = b_2$ $\equiv$unsatisfiable	$\neg (\exists i : q_i = b_0 \vee \exists i : q_i = b_1$ $\Rightarrow \exists i : q_i = b_0 \vee \exists i : q_i = b_1)$ $\equiv$unsatisfiable

by hand. To illustrate the safety and fixed-point checks, consider a nondeterministic finite state automaton (NFA) with three states shown in Figure 3.6. Observe that from b_2 each of b_0 and b_1 are unreachable and vice versa. Let

$$\upsilon(\mathcal{N}) \triangleq \exists i \in [\mathcal{N}] : q[i] = b_0, \text{ and } \mathsf{Init}_i \triangleq \forall i \in [\mathcal{N}] : q[i] = b_2,$$

so that the illegal state b_0 is unreachable from the initial state. The initial states are unreachable, which we will illustrate unsatisfiability in the safety check $\rho_k \wedge \mathsf{Init}_i$. As there are three transitions in each NFA, there are three rules describing the transitions in the parameterized network. The individual transition rules are specified in Figure 3.6. The backwards reachability algorithm from Figure 3.5 with safety and fixed-point checks for the FSA example is executed and shown in Table 3.1. We replace the bracket notation $[i]$ with underscore notation $_i$ in the FSA example for brevity. We can see that the safety check,

$\rho_k \wedge \mathsf{Init}_i$ is unsatisfiable for all k. Since the fixed-point check, $\neg(\rho_k \Rightarrow \rho_{k-1})$ is unsatisfiable for $k = 2$, the algorithm terminates and returns that the system is safe.

3.5 SATS Properties Verified

We specify and verify several of the same safety properties verified for SATS using PVS [27, 68]. We leave out properties regarding lateral entry that we are not modeling. The initial states are specified as the formula:

$$\mathsf{Init}_i \triangleq \forall i \in \mathcal{N} : q[i] = \mathbf{fly} \wedge m[i] \in \{left, right\} \wedge x[i] = 0 \wedge s[i] = 0 \wedge c = 0.$$

Observe that all of the properties verified are in essence mutual exclusion properties. Some properties state that are no more than a single aircraft in a state, while others specify no more than two are in a state, etc. Separation assurance can be thought of as a sort of physical mutual exclusion property.

The safety properties are specified as index-quantified *Passel* assertions.

(A) There are no more than four aircraft attempting to land, that is, the total number of aircraft in any states besides flying and landed is four (but there may be arbitrarily many aircraft flying or landed). Let $F = \mathsf{L} \setminus \{\mathbf{fly}, \mathbf{taxi}\}$ be the set of discrete locations without the flying or taxi states, then the property is specified as:

$$\begin{aligned}\zeta(\mathcal{N}) \triangleq \dot{\forall} i, j, k, l, m \in [\mathcal{N}] : (q[i] \in F \wedge q[j] \in F \wedge q[k] \in F \wedge q[l] \in F) \\ \Rightarrow q[m] \in \{\mathbf{fly}, \mathbf{taxi}\},\end{aligned}$$

where we recall $\dot{\forall}$ means all quantified variables are distinct (i.e., here $i \neq j \neq k \neq l \neq m$). In the MCMT model, we actually use counters to track this property, e.g., we count the number of aircraft in the system and verify this counter is bounded from above by four. It is prohibitively expensive to use too many index variables in a formula. We note that the SATS specification allows only four aircraft to be on approach at a given time [25, 26], but the number of aircraft involved in the protocol could potentially be

expanded by adding additional "sides" with corresponding holding, base, and missed zones (e.g., SATS is designed to have only left and right sides, but one can imagine a system with more entry points).

(B) The main property we are interested in is separation assurance, that two aircraft on approach to the runway are separated by a safety spacing $L_S > 0$. Formally, the property is:

$$\zeta(\mathcal{N}) \triangleq \dot{\forall} i, j \in [\mathcal{N}] : \big(q[i] \in \{\mathbf{b^r}, \mathbf{b^l}, \mathbf{fin}, \mathbf{m^r}, \mathbf{m^l}\} \wedge q[j] \in \{\mathbf{b^r}, \mathbf{b^l}, \mathbf{fin}, \mathbf{m^r}, \mathbf{m^l}\} \\ \wedge s[i] = s[j] - 1) \Rightarrow x[i] - x[j] \geq L_S.$$

(C) No more than two aircraft are actually on either side (left or right). The property for the left side (and symmetrically, right) is:

$$\zeta(\mathcal{N}) \triangleq \dot{\forall} i, j, k \in [\mathcal{N}] : (q[i] \in \{\mathbf{h3^l}, \mathbf{h2^l}, \mathbf{b^l}, \mathbf{m^l}\} \wedge q[j] \in \{\mathbf{h3^l}, \mathbf{h2^l}, \mathbf{b^l}, \mathbf{m^l}\}) \\ \Rightarrow q[k] \in \mathsf{L} \setminus \{\mathbf{h3^l}, \mathbf{h2^l}, \mathbf{b^l}, \mathbf{m^l}\}.$$

(D) At most one aircraft is in each of the holding zones, for $\mathbf{h2^r}$ (and defined analogously for $\mathbf{h3^r}$, $\mathbf{h3^l}$, and $\mathbf{h2^l}$) this is:

$$\zeta(\mathcal{N}) \triangleq \dot{\forall} i, j \in [\mathcal{N}] : (q[i] = \mathbf{h2^r}) \Rightarrow q[j] \neq \mathbf{h2^r}.$$

We could not verify the property for $\mathbf{h3^l}$ or $\mathbf{h3^r}$ due to a spurious execution from the stopping failures abstraction [54], so we assumed these two cases, and were able to establish the property for $\mathbf{h2^l}$ and $\mathbf{h2^r}$.

(E) No more than two aircraft are on a missed approach fix, for the right (and defined analogously for left), this is:

$$\zeta(\mathcal{N}) \triangleq \dot{\forall} i, j, k \in [\mathcal{N}] : (q[i] = \mathbf{m^r} \wedge q[j] = \mathbf{m^r}) \Rightarrow q_k \neq \mathbf{m^r}.$$

Additionally, there is a liveness property proven in [27], but MCMT cannot verify liveness

properties, only safety ones. The property would state that all aircraft eventually land and that they land in order according to their sequence numbers. Some very recent work attempts to allow verification of some classes of liveness properties [33], but general liveness properties for parameterized timed systems were shown to be undecidable in [47].

Experimental Setup for MCMT. We use version 1.1.1 of MCMT, which uses version 1.0.32 of the SMT solver Yices [147]. MCMT has some capability to generate invariants, and we enabled the full invariant search for our verification (we used the options `-I` and `-S3`). All runtimes of verification attempts are reported in Table 3.2, and are measured on a modern laptop with 8 GB main memory and an Intel Core i7 quad-core processor running at 2.0 GHz. However, we ran the verification in a virtual machine under Ubuntu, limited to the use of two cores and 1.5 GB memory. We use the `memtime` utility from UPAAAL [38] to measure runtimes and memory usage. We use an existing model of the system in UPAAAL to verify properties of finite instantiations $\mathcal{A}^{\mathsf{N}}$ for $\mathsf{N} \in \{2, 3, 4, 5\}$ aircraft. We do not report verification runtimes for UPAAAL, as we primarily use UPAAAL as a simulation tool prior to encoding the SATS protocol in MCMT, but we could not verify beyond five aircraft in the system.

The order in which the properties are proved is important, as our process was first to attempt proving a property, and if it was established as an invariant, we would assume it as a lemma and continue the verification process. We used a Python script to automatically call MCMT and assert lemmas. It may seem surprising that the total number of aircraft property (A) takes a short amount of runtime. The intuition behind this was established in the manual inductive invariant proofs done in [68]: property (A) is itself an inductive invariant (Definition 2.4) without refinement. Finally, we note that we also perform some sanity checks to see if certain states are reachable. For example, we show there are three aircraft allowed in the system, although this could be spurious due to the stopping failures abstraction used by MCMT [148].

Table 3.2: Runtime in seconds and maximum memory usage in megabytes required to verify properties of SATS using MCMT.

Property	Runtime (s)	Memory Usage (MB)
(A)	25.95	10.19
(B)	283.08	32.49
(C)	24.50	5.80
(D)	0.81	4.56
(E)	491.61	274.44

3.6 Summary

In this chapter, we describe a backward reachability method for uniform verification of safety properties for parameterized networks of timed automata. Using this method as implemented in MCMT, we automatically verify several safety properties of a distributed air traffic landing protocol, regardless of the number of aircraft involved in the protocol. One could argue that in physical scenarios there are physical and geometric restrictions preventing an arbitrary number of cars or aircraft from interacting, and that instead at most a fixed, finite natural number N may interact. While this is true, uniform verification may enable scalable verification, as it may be infeasible to perform the composition of N such $\mathcal{A}(i)$ due to the growth in the size of the composed $\mathcal{A}^{\mathsf{N}}$. In particular, we found that using the approach implemented in model checkers like UPAAAL, we could not scale beyond a few aircraft. A small network $\mathcal{A}^{\mathsf{N}}$ may suffice for verification, but what if the number N that can feasibly be verified is smaller than the number one may expect in the network?

The air traffic protocol is a nontrivial distributed cyber-physical system (DCPS), and for this reason, we believe it could serve as a standard benchmark in the verification of DCPS. We believe other DCPS under development, like networks of autonomous vehicles or medical devices, can and should be verified using this approach to increase the assurance of reliability that the complex interaction of software and physical processes does not yield catastrophic failure for some instantiations of the system. While we were successful in the verification in this chapter—in part because there exists a cutoff on the number of aircraft in the system as shown through the verification approach used in [27]—it would be

interesting to investigate abstractions for solving this problem. For instance, the environment abstraction approach [80] tracks the number of processes satisfying some predicates, perhaps even abstracting the continuous variables in this way, and may provide a more tractable approach for some classes of DCPS.

Chapter 4

Reachability Using Anonymized States for Finite Instantiations of Hybrid Automata Networks

In this chapter, we present a method for computing the set of reachable states of *finite instantiations* of parameterized networks of hybrid automata. The method utilizes a symmetry-reduced representation of the set of reachable states (modulo the automata indices), which makes it scalable. Rather than explicitly enumerating all the automaton indices in formulas representing states, the symmetry-reduced representation tracks only: (a) the classes of automata, which are the states of automata represented with formulas over symbolic indices, and (b) the number of automata in each of the classes. We present an algorithm for overapproximating the reachable states by computing state transitions in this symmetry-reduced representation.

4.1 Introduction

While, in general, computing reachable sets of finite instantiations cannot be used for uniform verification of parameterized networks of hybrid automata—that is, verification of properties of the network $\mathcal{A}^{\mathcal{N}}$ for any $\mathcal{N} \in \mathbb{N}$ (see Definition 2.5)—it may be useful in methods for performing such verification. For example, such reachability computations are used as subroutines in the methods for synthesizing quantified candidate inductive invariants of Chapter 6. In addition, a designer may want to verify finite instantiations prior to verifying the general parameterized network, which is usually harder. Symmetry has long been studied in model checking, as it is one approach to alleviate the state-space explosion problem [114–118, 120, 124–126].

Satisfiability-modulo-theories (SMT) solvers generalize the classical Boolean satisfiability (SAT) problem to allow for constants of additional sorts, like reals, arrays, and integers.

Many solvers also have support for quantified theories, allowing for quantified sentences, such as the quantified theory of real linear arithmetic (RLA). Recent advances in quantified Boolean formula (QBF) solvers—often integrated within SMT solvers—have enabled QBF formulations of the bounded model checking problem, instead of the usual SAT formulation [149]. Such formulations allow for encoding in SMT solvers like Z3 [136, 150], as we do in this chapter. Bounded model checking using SMT solvers has been successfully used in analyzing timed systems for some time [151, 152]. The current state-of-the-art in using SMT solvers for BMC of timed systems is outlined in [153–155]. SMT-based reachability techniques have also been developed for hybrid systems with fairly general dynamics. For instance, [155] allows rectangular differential inclusions ($\dot{x} \in [a, b]$), and the SAT-modulo-ODE methods allow general dynamics [156–158].

Outline. Section 4.2 describes the anonymized (symmetry-reduced) state-space representation. Section 4.3 describes an on-the-fly algorithm for computing the reachable states in the anonymized representation. Section 4.4 analyzes the reachability algorithm and establishes its soundness. Section 4.4.1 gives an example with anonymized reachable states that are independent of the choice of N. Section 4.5 summarizes the anonymized state-space representation, reachability algorithm, and results. The experimental results using the reachability algorithm with anonymized states implemented in *Passel* appear in Chapter 7, Section 7.6

4.2 Anonymized State-Space Representation

Consider a hybrid automaton template $\mathcal{A}(\mathcal{N}, i)$ for some $i \in [\mathcal{N}]$. For any fixed $\mathsf{N} \in \mathbb{N}$, the composed automaton modeling a network of size N is $\mathcal{A}^{\mathsf{N}}$ and it is defined by Definition 2.2. Throughout this chapter, we fix $\mathcal{A}^{\mathsf{N}}$ and present an algorithm for computing $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$ that takes advantage of the symmetries in its hybrid automaton template $\mathcal{A}(i)$. More specifically, we present an efficient representation of $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$ that is *anonymized*, so numerical automaton indices—1, 2, …, N—are not explicitly enumerated and are instead modeled using symbolic indices—i_1, i_2, …, i_{N}.

Recall from Definition 2.2 that the state-space of $\mathcal{A}^{\mathsf{N}}$ is Q^{N}. For illustrating the anonymized

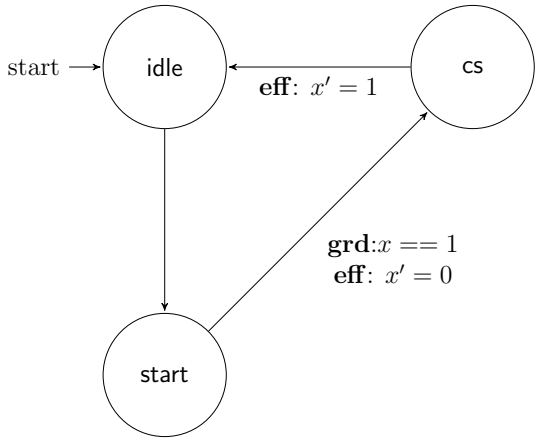

Figure 4.1: MUX-SEM mutual exclusion algorithm for automaton $\mathcal{A}(i)$ for illustrating the anonymized state-space representation.

```
automaton name='MUX-SEM'
  variable name='q[i]' type='L'     // location local variable
  variable name='x' type='boolean' // global mutex variable

  location name='idle'
  location name='start'
  location name='cs'

  transition from='idle' to='start'
  transition from='start' to='cs'
    grd: x = 1
    eff: x' = 0
  transition from='cs' to='idle'
    eff: x' = 0

property: forall i, j (i != j and q[i] = cs) implies (q[j] != cs)
initially: forall i (q[i] = idle and x = 1)
```

Figure 4.2: *Passel* input file specifying automaton template $\mathcal{A}(i)$ for mutual exclusion algorithm MUX-SEM.

representation of the state-space, we use the MUX-SEM mutual exclusion example shown graphically in Figure 4.1 and as a *Passel* specification in Figure 4.2. MUX-SEM has one local variable $q[i]$ with type $\mathsf{Loc} \triangleq \{\mathsf{idle}, \mathsf{start}, \mathsf{cs}\}$ and one global variable x of Boolean $\mathbb{B}$ type.[1] For $\mathsf{N} = 3$, the product of the types is $\mathsf{Loc}^3 \times \mathbb{B}$ which has $2\left|\mathsf{Loc}^3\right| = 54$ elements, which is the number of elements in the state-space of $\mathcal{A}^3$. For $\mathcal{A}^3$, the reachable states are

[1]While the *Passel* language defined in Section 2.3.2 does not allow for Boolean-typed variables, we model them as bounded integer-typed variables. That is, a variable specified with a Boolean type is syntactic sugar for an integer variable with an axiom that it takes values 0 or 1 only. Variables of bitvector type with length b are modeled as integers with axioms specifying they range 0 through 2^{b-1}.

encoded as the DNF formula:

$$(q[1] = \mathsf{idle} \wedge q[2] = \mathsf{idle} \wedge q[3] = \mathsf{idle} \wedge x = 1) \vee \tag{4.1}$$

$$(q[1] = \mathsf{start} \wedge q[2] = \mathsf{idle} \wedge q[3] = \mathsf{idle} \wedge x = 1) \vee \tag{4.2}$$

$$(q[1] = \mathsf{idle} \wedge q[2] = \mathsf{start} \wedge q[3] = \mathsf{idle} \wedge x = 1) \vee \tag{4.3}$$

$$(q[1] = \mathsf{idle} \wedge q[2] = \mathsf{idle} \wedge q[3] = \mathsf{start} \wedge x = 1) \vee \tag{4.4}$$

$$(q[1] = \mathsf{start} \wedge q[2] = \mathsf{start} \wedge q[3] = \mathsf{idle} \wedge x = 1) \ \vee \ (q[1] = \mathsf{start} \wedge q[2] = \mathsf{idle} \wedge q[3] = \mathsf{start} \wedge x = 1) \vee$$
$$(q[1] = \mathsf{idle} \wedge q[2] = \mathsf{start} \wedge q[3] = \mathsf{start} \wedge x = 1) \ \vee \ (q[1] = \mathsf{start} \wedge q[2] = \mathsf{start} \wedge q[3] = \mathsf{start} \wedge x = 1) \vee$$
$$(q[1] = \mathsf{cs} \wedge q[2] = \mathsf{idle} \wedge q[3] = \mathsf{idle} \wedge x = 0) \ \vee \ (q[1] = \mathsf{idle} \wedge q[2] = \mathsf{cs} \wedge q[3] = \mathsf{idle} \wedge x = 0) \vee$$
$$(q[1] = \mathsf{idle} \wedge q[2] = \mathsf{idle} \wedge q[3] = \mathsf{cs} \wedge x = 0) \ \vee \ (q[1] = \mathsf{cs} \wedge q[2] = \mathsf{start} \wedge q[3] = \mathsf{idle} \wedge x = 0) \vee$$
$$(q[1] = \mathsf{start} \wedge q[2] = \mathsf{cs} \wedge q[3] = \mathsf{idle} \wedge x = 0) \ \vee \ (q[1] = \mathsf{start} \wedge q[2] = \mathsf{idle} \wedge q[3] = \mathsf{cs} \wedge x = 0) \vee$$
$$(q[1] = \mathsf{cs} \wedge q[2] = \mathsf{start} \wedge q[3] = \mathsf{start} \wedge x = 0) \ \vee \ (q[1] = \mathsf{start} \wedge q[2] = \mathsf{cs} \wedge q[3] = \mathsf{start} \wedge x = 0) \vee$$
$$(q[1] = \mathsf{start} \wedge q[2] = \mathsf{start} \wedge q[3] = \mathsf{cs} \wedge x = 0) \ \vee \ (q[1] = \mathsf{cs} \wedge q[2] = \mathsf{idle} \wedge q[3] = \mathsf{start} \wedge x = 0) \vee$$
$$(q[1] = \mathsf{idle} \wedge q[2] = \mathsf{cs} \wedge q[3] = \mathsf{start} \wedge x = 0) \ \vee \ (q[1] = \mathsf{idle} \wedge q[2] = \mathsf{start} \wedge q[3] = \mathsf{cs} \wedge x = 0). \tag{4.5}$$

This DNF representation has 20 clauses (conjunctive formulas), where each clause represents one state $\mathbf{x} \in \mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$. This representation is inefficient because it explicitly enumerates all the permutations of automata indices, and does not exploit the fact that several of the reachable states are equivalent modulo the automaton indices. For a given state $\mathbf{x} \in Q^{\mathsf{N}}$, the set of corresponding states $\mathbf{X} \subseteq Q^{\mathsf{N}}$ that are equivalent modulo indices is obtained by substituting any index i of all local variables $v[i] \in \mathsf{V}_L[i]$ with an index $j \in [\mathsf{N}]$.

Definition 4.1 *Two states* $\mathbf{x}, \mathbf{x}' \in Q^{\mathsf{N}}$ *of* $\mathcal{A}^{\mathsf{N}}$, *are* equivalent modulo indices *if there exists a bijection* $\pi : [\mathsf{N}] \to [\mathsf{N}]$ *such that for each* $v[i] \in \mathsf{V}_i$, $\mathbf{x}.v[i] = \mathbf{x}'.v[\pi(i)]$. *For a state* $\mathbf{x} \in Q^{\mathsf{N}}$ *of* $\mathcal{A}^{\mathsf{N}}$, *the set of states* $\mathbf{X}'$ *that is equivalent modulo indices to* $\mathbf{x}$ *is the set*

$$\{\mathbf{x}' \in Q^{\mathsf{N}} \mid \mathbf{x} \text{ and } \mathbf{x}' \text{ are equivalent modulo indices}\}.$$

This is the same type of definition as the existence of an automorphism used in [114–116], but we do not add this additional structure as we will not utilize tools from group theory.

For the MUX-SEM example with a state $\mathbf{x}$ satisfying $(q[1] = \mathsf{start} \wedge q[2] = \mathsf{idle} \wedge q[3] = \mathsf{idle} \wedge x = 1)$ (Equation 4.2), the states equivalent modulo indices to $\mathbf{x}$ are those satisfying

any of the following formulas:

$$(q[1] = \mathsf{start} \wedge q[2] = \mathsf{idle} \wedge q[3] = \mathsf{idle} \wedge x = 1), \text{ or} \tag{4.6}$$

$$(q[1] = \mathsf{idle} \wedge q[2] = \mathsf{idle} \wedge q[3] = \mathsf{start} \wedge x = 1), \text{ or} \tag{4.7}$$

$$(q[1] = \mathsf{idle} \wedge q[2] = \mathsf{start} \wedge q[3] = \mathsf{idle} \wedge x = 1), \tag{4.8}$$

which respectively correspond to Equations 4.2, 4.3, and 4.4. Of course, a state is equivalent modulo indices to itself by picking the bijection π to be the identity mapping (for example, as in Equation 4.6). For Equations 4.7 and 4.8, we can explicitly define a bijection π as:

$$\pi(i) = (i \mod N) + 1.$$

For a formula ϕ, we will overload π and write $\pi(\phi)$, which modifies ϕ by applying π to each index variable i in ϕ. Continuing with the MUX-SEM example, applying this π to $(q[1] = \mathsf{start} \wedge q[2] = \mathsf{idle} \wedge q[3] = \mathsf{idle} \wedge x = 1)$ (Equation 4.2) yields:

$$\begin{aligned}
&\pi((q[1] = \mathsf{start} \wedge q[2] = \mathsf{idle} \wedge q[3] = \mathsf{idle} \wedge x = 1)) \\
&= (q[2] = \mathsf{start} \wedge q[3] = \mathsf{idle} \wedge q[1] = \mathsf{idle} \wedge x = 1) \\
&= (q[1] = \mathsf{idle} \wedge q[2] = \mathsf{start} \wedge q[3] = \mathsf{idle} \wedge x = 1) \text{ (equals Equation 4.3).}
\end{aligned}$$

The anonymized state representation takes this idea a step further by utilizing symbolic names for process indices along with counters, and a formula representing the valuations of any global variables. Consider Equations 4.2, 4.3, and 4.4. An equivalent description of the local variable valuations is to say there is one automaton in start and two automata in idle, without referring to which automaton is in which state. More explicitly, this set of states is the set of states where (a) there are two classes of automata, (b) one class has one automaton, say i, with $q[i] = \mathsf{start}$, (c) the other class has two automata, such that for each automaton i, $q[i] = \mathsf{idle}$, and (d) the global variable valuation satisfies $x = 1$. We now define this representation in general.

Definition 4.2 *For a fixed* $\mathsf{N} \in \mathbb{N}$ *and an automaton template* $\mathcal{A}(\mathsf{N}, i)$*, consider the au-*

tomaton network $\mathcal{A}^{\mathsf{N}}$ *(Definition 2.2). An* anonymized state representation $\mathtt{S}$ *is a tuple* $\langle \mathtt{Classes}, \mathtt{G} \rangle$*, where:*

(a) Each anonymized class $\mathtt{C} \in \mathtt{Classes}$ *is a tuple* $\mathtt{C} \triangleq \langle \mathtt{Count}, \mathtt{I}, \mathtt{Form} \rangle$*, where:*

(i) $\mathtt{I}$ *is a natural number called the class's rank, which is equal to the number of distinct symbolic index variables appearing in* $\mathtt{Form}$*:* $\mathtt{I} = |ivars(\mathtt{Form})|$.

(ii) $\mathtt{Form}$ *is a quantifier-free Passel assertion over the variables* $\mathsf{V}_L[i_1] \cup \ldots \cup \mathsf{V}_L[i_{\mathtt{C.I}}]$*, where* i_1*, ...,* $i_{\mathtt{I}}$ *are* $\mathtt{I}$ *distinct symbolic index variables.*

(iii) $\mathtt{Count}$ *is a natural number called the class's count, and satisfies* $\mathsf{N} \geq \mathtt{Count} \geq |\mathtt{I}|$*. The count is the number of automata of class* $\mathtt{C}$*.*

Additionally, the sum of all the class counts in $\mathtt{S}$ *equals* N*:*

$$\mathsf{N} = \sum_{\mathtt{C} \in \mathtt{S.Classes}} \mathtt{C.Count},$$

where $\mathtt{C.Count}$ *is the count of class* $\mathtt{C}$*.*

(b) $\mathtt{G}$ *is a quantifier-free Passel assertion over the global variables* V_G*.*

When necessary to refer to various components of an anonymized state representation $\mathtt{S}$, we write:

$$\mathtt{S} = \left\langle \left\{ \overbrace{\underbrace{\langle \mathtt{Count}_1, \mathtt{I}_1, \mathtt{Form}_1 \rangle}_{\mathtt{C}_1}, \underbrace{\langle \mathtt{Count}_2, \mathtt{I}_2, \mathtt{Form}_2 \rangle}_{\mathtt{C}_2}, \ldots}^{\mathtt{Classes}} \right\}, \mathtt{G} \right\rangle.$$

We use the (.) notation to refer to particular elements of tuples. For example, $\mathtt{C.Count}$ refers to the count of a particular anonymized class $\mathtt{C}$, and likewise, $\mathtt{C.Form}$ refers to its formula. When the number of index variables $\mathtt{C.I}$ is clear from context, we drop it from the $\mathtt{C}$ tuple and write $\langle \mathtt{Count}, \mathtt{Form} \rangle$.

We say two anonymized classes $\mathtt{C}_1$ and $\mathtt{C}_2$ are equivalent and write $\mathtt{C}_1 \equiv \mathtt{C}_2$ iff they have the same class formulas, class counts, and class variable counts:

$$\mathtt{C}_1.\mathtt{Count} = \mathtt{C}_2.\mathtt{Count} \wedge \mathtt{C}_1.\mathtt{I} = \mathtt{C}_2.\mathtt{I} \wedge \mathtt{C}_1.\mathtt{Form} \Leftrightarrow \mathtt{C}_2.\mathtt{Form}. \tag{4.9}$$

Here, equivalence between the class formulas is a semantic and not syntactic notion, and means the formula $\mathtt{C}_1.\mathtt{Form} \Leftrightarrow \mathtt{C}_2.\mathtt{Form}$ is valid (Section 2.4.2, Definition 2.3). We say two anonymized state representations $\mathtt{S}_1$ and $\mathtt{S}_2$ are equivalent and write $\mathtt{S}_1 \equiv \mathtt{S}_2$ iff they have the same state counts, the classes in their sets of classes are equivalent, and their global formulas are equivalent:

$$\forall\ \mathtt{C}_1 \in \mathtt{S}_1.\mathtt{Classes}\ \exists\ \mathtt{C}_2 \in \mathtt{S}_2.\mathtt{Classes}\ \mathtt{C}_1 \equiv \mathtt{C}_2 \wedge \mathtt{G}_1 \equiv \mathtt{G}_2.$$

For an anonymized class $\mathtt{C}$, the requirement of Definition 4.2 that $\mathtt{C.Count} \geq |\mathtt{C.I}|$ means the number of automata satisfying the $\mathtt{Form}$ is at least as large as the number of distinct index variables appearing in $\mathtt{Form}$. For example, this requirement means:

$$\langle 2, 2, q[i] = \mathsf{cs} \wedge q[j] = \mathsf{idle} \rangle\ ,\ \text{is an anonymized class, but}$$
$$\langle 1, 2, q[i] = \mathsf{cs} \wedge q[j] = \mathsf{idle} \rangle\ ,\ \text{is not an anonymized class.}$$

This also restricts class counts of zero.

The interpretation of an anonymized state representation corresponds to a set of states of Q^{N}, which is written as $[\![\langle \mathtt{Classes}, \mathtt{G} \rangle]\!]$. Since the class formulas of $\mathtt{S}$ are over the variables of automata with symbolic indices, the interpretation instantiates the symbolic indices with specific elements of $[\mathsf{N}]$, which yields the set of states that are equivalent modulo indices.

Definition 4.3 *For an anonymized state representation*

$$\mathtt{S} = \left\langle \left\{ \underbrace{\langle \mathtt{Count}_1, \mathtt{I}_1, \mathtt{Form}_1 \rangle}_{\mathtt{C}_1}, \ldots, \underbrace{\langle \mathtt{Count}_k, \mathtt{I}_k, \mathtt{Form}_k \rangle}_{\mathtt{C}_k} \right\}, \mathtt{G} \right\rangle,$$

we assign all possible values in $[\mathsf{N}]$ *to the set of symbolic index variables* $i_1, \ldots, i_{\mathtt{I}_k}$ *as follows. Let*

$$p = \left\{ \underbrace{\{p_1^1, \ldots, p_1^{\mathtt{I}_1}\}}_{p_1}, \ldots, \underbrace{\{p_k^1, \ldots, p_k^{\mathtt{I}_k}\}}_{p_k} \right\}$$

be a partition of $[\mathsf{N}]$*, such that, for any* $p_j \subseteq p$*,*

(a) $|p_j| = \mathtt{Count}_j$ *and*

(b) p_j *is partitioned into* $\mathtt{I}_j$ *sets* $p_j^1, \ldots, p_j^{\mathtt{I}_j}$*, where we recall that* $\mathtt{I}_j$ *is the rank of* $\mathtt{C}_j$*.*

We note that (a) $\sum_{p_j \in p} |p_j| = \mathsf{N}$ *since* p *partitions* $[\mathsf{N}]$*, and (b)* $\mathtt{Count}_j \geq \mathtt{I}_j$ *(by Definition 4.2, (iii)). Then, the set of states of network* $\mathcal{A}^{\mathsf{N}}$ *represented by* $\mathtt{S}$ *corresponding to partition* p *are:*

$$[\![\mathtt{S}_p]\!] \triangleq \{\mathbf{x} \in Q^{\mathsf{N}} \mid \mathbf{x} \models \mathtt{G} \wedge \mathtt{Form}_1(p_1) \wedge \ldots \mathtt{Form}_k(p_k)\}, \tag{4.10}$$

where each $\mathtt{Form}_j(p_j)$ *is:*

$$\mathtt{Form}_j(p_j) \triangleq \forall i_j^1 \in p_j^1, \ldots, i_j^{\mathtt{I}_j} \in p_j^{\mathtt{I}_j} \; : \; \mathtt{Form}_j(i_j^1, \ldots, i_j^{\mathtt{I}_j}). \tag{4.11}$$

We have written $\mathtt{Form}_j(i_j^1, \ldots, i_j^{\mathtt{I}_j})$ *to highlight that* $\mathtt{Form}_j$ *is over* $\mathtt{I}_j$ *symbolic index variables. Note that* $\mathtt{Form}_j(p_j)$ *is equivalent to a finite-length conjunction since each* $p_j^{\mathtt{I}_j}$ *is finite. The complete set of states of network* $\mathcal{A}^{\mathsf{N}}$ *represented by* $\mathtt{S}$ *are:*

$$\bigcup_p [\![\mathtt{S}_p]\!] \text{ for any partition } p \text{ respecting Equation 4.10.} \tag{4.12}$$

For MUX-SEM, the anonymized state $\langle \{\langle 1, 1, q[i] = \mathsf{start}\rangle, \langle 2, 1, q[i] = \mathsf{idle}\rangle\}, x = 1\rangle$ corresponds to the following set of states of Q^{N}:

$$\begin{aligned} &[\![\langle \{\langle 1, 1, q[i] = \mathsf{start}\rangle, \langle 2, 1, q[i] = \mathsf{idle}\rangle\}, x = 1\rangle]\!] \\ = \; &\{\mathbf{x} \in Q^3 \mid \mathbf{x} \models (q[1] = \mathsf{start} \wedge q[2] = \mathsf{idle} \wedge q[3] = \mathsf{idle} \wedge x = 1) \vee \\ &(q[1] = \mathsf{idle} \wedge q[2] = \mathsf{start} \wedge q[3] = \mathsf{idle} \wedge x = 1) \vee \\ &(q[1] = \mathsf{idle} \wedge q[2] = \mathsf{idle} \wedge q[3] = \mathsf{start} \wedge x = 1) \; \}, \end{aligned}$$

where the formulas correspond respectively to Equations 4.2, 4.3, and 4.4. Consider the following anonymized representation with count three and rank two:

$$[\![\langle \{\langle 3, 2, q[i_1] = \mathsf{base} \wedge q[i_2] = \mathsf{base} \wedge x[i_2] \geq x[i_1] + L_S\rangle\}, \mathit{true}\rangle]\!].$$

One particular allowed partition is:

$$p = \left\{ \underbrace{\{1\}}_{p_1^1}, \underbrace{\{2,3\}}_{p_1^2} \right\}.$$

For this p, the states represented by $\mathtt{S}_p$ are:

$$\begin{aligned}
[\![\mathtt{S}_p]\!] = \; & \{\mathbf{x} \in Q^3 \mid \mathbf{x} \models \forall i_1^1 \in p_1^1, i_1^2 \in p_1^2 \; : \; q[i_1] = \mathsf{base} \wedge q[i_2] = \mathsf{base} \wedge x[i_2] \geq x[i_1] + L_S\} \\
= \; & \{\mathbf{x} \in Q^3 \mid \mathbf{x} \models (q[1] = \mathsf{base} \wedge q[2] = \mathsf{base} \wedge q[3] = \mathsf{base} \wedge \\
& \qquad x[2] \geq x[1] + L_S \wedge x[3] \geq x[1] + L_S) \; \}.
\end{aligned}$$

Note that $\{\{1,2,3\}\}$ is not an allowed partition since it is partitioned into one set, but $\mathtt{I} = 2$, and Definition 4.3 requires each $p_j \in p$ be partitioned into $\mathtt{I}_j$ partitions. For instance, the following are the other allowed partitions: $\{\{2\},\{1,3\}\}$, $\{\{3\},\{1,2\}\}$, $\{\{1,2\},\{3\}\}$, $\{\{1,3\},\{2\}\}$, and $\{\{2,3\},\{1\}\}$. All these allowed partitions define the full set of states $[\![\mathtt{S}]\!]$ that the anonymized state $\mathtt{S}$ represents. We note that this is equivalent to all the states equivalent modulo indices to the states $[\![\mathtt{S}_p]\!]$ for a particular partition p.

Every set of states in Q^{N} has a finite representation as an anonymized state representation. This can be seen by the following. Sets of states in Q^{N} may be represented as quantifier-free *Passel* assertions, so consider such an assertion $\phi(1,\ldots,\mathsf{N})$, where we explicitly reference the numerical indices used in the assertion. We can create an anonymized state representation $\mathcal{S}$ with a single class $\mathtt{C}$ as follows. Let $\mathtt{C.Count} = \mathsf{N}$ and $\mathtt{C.Form} = \phi(i_1,\ldots,i_{\mathsf{N}})$, where $\phi(i_1,\ldots,i_{\mathsf{N}})$ is $\phi(1,\ldots,\mathsf{N})$ replacing the numerical indices with symbolic ones. Of course, this does not make the anonymized representation of the set of states more efficient since $\phi(i_1,\ldots,i_{\mathsf{N}})$ is the same size as $\phi(1,\ldots,\mathsf{N})$.

The set of reachable states of MUX-SEM represented using anonymized state representa-

tions (Definition 4.2) for MUX-SEM for $\mathsf{N} = 3$ is:

$$\begin{aligned}
&\langle\{\langle 3, q[i] = \mathsf{idle}\rangle\}, x = 1\rangle, \\
&\langle\{\langle 1, q[i] = \mathsf{start}\rangle, \langle 2, q[i] = \mathsf{idle}\rangle\}, x = 1\rangle, \\
&\langle\{\langle 2, q[i] = \mathsf{start}\rangle, \langle 1, q[i] = \mathsf{idle}\rangle\}, x = 1\rangle, \\
&\langle\{\langle 3, q[i] = \mathsf{start}\rangle\}, x = 1\rangle, \\
&\langle\{\langle 1, q[i] = \mathsf{cs}\rangle, \langle 2, q[i] = \mathsf{idle}\rangle\}, x = 0\rangle, \\
&\langle\{\langle 1, q[i] = \mathsf{cs}\rangle, \langle 1, q[i] = \mathsf{idle}\rangle, \langle 1, q[i] = \mathsf{start}\rangle\}, x = 0\rangle, \\
&\langle\{\langle 1, q[i] = \mathsf{cs}\rangle, \langle 2, q[i] = \mathsf{start}\rangle\}, x = 0\rangle.
\end{aligned} \tag{4.13}$$

This representation of reachable states has seven elements.

The reachable states of $\mathcal{A}^{\mathsf{N}}$ represented as anonymized states is denoted by AnonReach, and is:

$$\mathsf{AnonReach} \triangleq \{\mathtt{S} \mid \mathbf{x} \in \mathsf{Reach}(\mathcal{A}^{\mathsf{N}}) \ \wedge \ \mathbf{x} \in [\![\mathtt{S}]\!]\}.$$

The set of reachable class formulas `ReachForms` for $\mathcal{A}^{\mathsf{N}}$ are all the quantifier-free *Passel* assertions appearing in any anonymized class of any state `S` in any anonymized reachable state AnonReach, conjuncted with the global formula `G` of `S`. That is:

$$\mathtt{ReachForms} \triangleq \{\mathtt{C.Form} \mid \mathbf{x} \in \mathsf{Reach}(\mathcal{A}^{\mathsf{N}}) \ \wedge \ \mathbf{x} \in [\![\mathtt{S}]\!] \ \wedge \ \mathtt{C} \in \mathtt{S.Classes}\}.$$

For the MUX-SEM example with $\mathsf{N} = 3$, the set of reachable anonymized class formulas `ReachForms` is:

$$\begin{aligned}
&\{q[i] = \mathsf{idle} \wedge x = 1, q[i] = \mathsf{start} \wedge x = 1, q[i] = \mathsf{cs} \wedge x = 0, \\
&\ q[i] = \mathsf{idle} \wedge x = 0, q[i] = \mathsf{start} \wedge x = 0\}.
\end{aligned} \tag{4.14}$$

Thus, for MUX-SEM there are a total of $|\mathtt{ReachForms}| = 5$ distinct formulas used to represent the reachable states. This highlights how the anonymized representation can enable

computation of reach sets for large N. The number of classes may remain constant beyond a certain N, at which point only one natural number per class is needed to represent larger and larger instances. For MUX-SEM, the five formulas of Equation 4.14 are enough to represent the reachable states for any network $\mathcal{A}^{\mathsf{N}}$ for $\mathsf{N} \geq 1$.

4.3 Reachability Using Anonymized States

Having illustrated the anonymized representation, we now describe an on-the-fly algorithm for computing the reachable states of finite instances of $\mathcal{A}^{\mathsf{N}}$ using this representation. Pseudocode for the algorithm appears in Figure 4.3. We first make some assumptions about the format of the class formulas.

Assumption 4.4 *For an anonymized state* $\mathtt{S}$*, for each class* $\mathtt{C} \in \mathtt{Classes}$*, we assume the class formula* $\mathtt{C.Form}$ *is in conjunctive normal form, and for each index* $i \in i_1, \ldots, i_{\mathtt{C.I}}$*,* $\mathtt{C.Form}$ *contains an equality* $q[i] = \mathsf{l}$ *for some location* $\mathsf{l} \in \mathsf{L}$*.*

This assumption ensures that all of the class formulas in the anonymized states correspond to convex sets and that each class has the control location specified, since the continuous dynamics are specified for each location in L (recall Definition 2.1). The CNF assumption is not restrictive: if a new class is created during the execution of the algorithm that contains disjunctions, then multiple classes with CNF formulas are created.

The algorithm inputs are: (a) a template hybrid automaton $\mathcal{A}(\mathcal{N}, i)$, (b) an initial condition assertion Init_i, and (c) a constant natural number N. The algorithm operates on frontiers of reachable states represented by the set Frontier. The set of reachable states computed so far is represented by the set AnonReach. At initialization (line 3), Frontier is set to be an anonymized state representation containing one anonymized class with a count equal to N and class formula equal to the body of the index-quantified *Passel* assertion over the local variables, $\mathsf{Init}_{\mathsf{L}}[i]$. The global assertion of the initial anonymized representation is initialized with $\mathsf{Init}_{\mathsf{G}}[i]$, which is the body of the index-quantified *Passel* assertion over the global

```
1  function symreach(𝒜(𝒩, i), Init_i, N) {
     AnonReach ← ∅
3    Frontier ← {⟨{⟨N, Init_L[i]⟩}, Init_G[i]⟩} // create initial anonymized state

5    // repeat until no new states are added to the frontier
     while Frontier ≠ ∅ {
7      Frontier_New ← ∅ // initialize next frontier
       AnonReach ← AnonReach ∪ Frontier // add frontier to reachable states
9      // compute successors of each anonymized state representation in the frontier
       foreach anonymized state S in Frontier {
11       Frontier_New ← Frontier_New ∪ discreteSuccessors(S, AnonReach, Frontier_New)

13       Frontier_New ← Frontier_New ∪ contSuccessors(S, AnonReach, Frontier_New)
       }
15     Frontier ← Frontier_New
     }
17 }
```

Figure 4.3: On-the-fly reachability algorithm using anonymized states. The input arguments are an automaton specification $\mathcal{A}(\mathcal{N}, i)$, an initial condition assertion Init_i, and a constant natural number N. The anonymized representation of the reachable states are computed as a fixed-point computation starting from the anonymized representation of the initial states. The algorithm computes the set of anonymized reachable states AnonReach.

variables. For example, for MUX-SEM, this initializes:

$$\mathsf{Frontier} = \{\langle\{\langle \mathsf{N}, 1, q[i] = \mathsf{idle}\rangle\}, x = 1\rangle\}.$$

Next, the main loop (line 6) removes an anonymized state representation S from Frontier, computes the successors of S, and continues until no new anonymized state representations are added to Frontier. New anonymized state representations are added to the frontier using the set $\mathsf{Frontier}_{\mathtt{New}}$ (line 7).[2] The set of anonymized reachable states is maintained by the set AnonReach (line 8).

The loop starting at line 10 computes the successors for each anonymized state representation S in Frontier, which are the states reachable from S in one step. It is composed of two parts: (a) computing the discrete successors corresponding to transitions (line 11), and (b) computing the continuous successors corresponding to trajectories (line 13).

[2]This description makes the algorithm a breadth-first traversal of the reachable states.

```
1   function discreteSuccessors(S, AnonReach, Frontier_New) {
      States_New ← ∅
3     foreach anonymized class C in S.Classes {
        foreach index i in {i_1, ..., i_C.I} {
5         foreach transition t in Trans_i {
            S_New ← post(t, S, C)
7           if S_New ∉ (AnonReach ∪ States_New ∪ Frontier_New) {
              States_New ← States_New ∪ {S_New}
9           }
          }
11      }
      }
13    return States_New
    }
```

Figure 4.4: Discrete successors of anonymized state representation S, which takes an anonymized state representation S, the current anonymized reachable states AnonReach, and the current new frontier $\mathsf{Frontier}_{\mathtt{New}}$, and returns a set of new anonymized state representations (if any), which are the discrete successors of the anonymized state representation S over all the transitions of $\mathcal{A}(\mathcal{N}, i)$.

4.3.1 Discrete Successors

The discrete successors are computed as shown in Figure 4.4. The function discreteSuccessors has three inputs: an anonymized state representation S, the anonymized reachable states computed thus far AnonReach, and the new frontier computed thus far $\mathsf{Frontier}_{\mathtt{New}}$. The output of discreteSuccessors are the anonymized states $\mathsf{States}_{\mathtt{New}}$ that may be reached from any class $\mathtt{C} \in \mathtt{S.Classes}$ of the state S, through any transition $\mathtt{t} \in \mathsf{Trans}_i$. These new states $\mathsf{States}_{\mathtt{New}}$ are added to the new frontier $\mathsf{Frontier}_{\mathtt{New}}$, then this repeats for any other states on the frontier Frontier from the previous iteration (see Figure 4.3, line 11).

The function begins with a loop iterating over each class C in the classes of the anonymized state representation S.Classes (line 3). There is a subsequent loop over each index variable i in the set of index variables in the class formula, $\{i_1, \ldots, i_{\mathtt{C.I}}\}$ (line 4). We continue with the MUX-SEM example for $\mathsf{N} = 3$ supposing $\mathtt{S} = \langle \{\langle 3, 1\rangle\, q[i] = \mathsf{idle}\}, x = 1\rangle$. For MUX-SEM, S.Classes has one element, so $\mathtt{C} = \langle 3, q[i] = \mathsf{idle}\rangle\, x = 1$ and the only index variable is i.

Next is an iteration over the (syntactic) transitions Trans_i of $\mathcal{A}(\mathcal{N}, i)$ (line 5). For each transition $\mathtt{t} \in \mathsf{Trans}_i$, the discrete post-states with respect to t from S by an automaton i with states satisfying C.Form, written $post(\mathtt{t}, \mathtt{S}, \mathtt{C})$, are computed next (Figure 4.5 called at line 6). Note that Trans_i has a dependence upon i.

The first step of of the discrete post is to compute the new class formula and count (line 3).

```
    function post(t, S, C) {
 2    // compute new class
      C_New.Form ← QuantElim(∃V_i : C.Form ∧ S.G ∧ grd(t, i) ∧ eff(t, i))
 4    // substitute primed variables with unprimed variables
      C_New.Form ← Substitute(C_New.Form, V'_i, V_i)
 6    // project away local variables (for global constraint)
      S_New.G ← QuantElim(∃V_i : C_New.Form)
 8    // project away global variables (for local constraint)
      C_New.Form ← QuantElim(∃V_G : C_New.Form)
10    C_New.Count ← 1
      // remove old class from new anonymized state representation classes
12    S_New.Classes ← S.Classes \ {C}
      // add old class to new anonymized state representation if its count will be positive
14    if C.Count ≠ 1 {
        S_New.Classes ← S.Classes ∪ {⟨C.Count − 1, C.Form⟩}
16    }
      // add new class to new anonymized state representation
18    S_New.Classes ← S_New.Classes ∪ {C_New}
      S_New ← mergeClasses(S_New)
20    return S_New
    }
```

Figure 4.5: Discrete post of an anonymized state representation $\mathtt{S}$ for an automaton with index i and states satisfying $\mathtt{C}$.

For a transition $\mathtt{t} \in \mathsf{Trans}_i$ and an anonymized class $\mathtt{C}$, the quantifier elimination step of line 3 computes the subsequent class from $\mathtt{C}$ under the transition $\mathtt{t}$, made by the automaton with index i.

Recall that $\mathbf{grd}(\mathtt{t})$ is a *Passel* assertion over V_i, and $\mathbf{eff}(\mathtt{t})$ is a *Passel* assertion over $\mathsf{V}_i \cup \mathsf{V}'_i$, where V'_i contains primed versions of each variable in V_i.[3] This computation can be carried out using quantifier elimination procedures over the types of the variables appearing in the guard, and effect of the transition $\mathtt{t}$, and then syntactically unpriming all primed variables (representing successors) following quantifier elimination (line 5). This step is an overapproximation, since it is computing the successors of each class regardless of the number of automata with states satisfying the anonymized class formula `Form`, and just presuming *there is some* automaton with variable valuations satisfying `Form`.

For the MUX-SEM example, we have $\mathtt{C} = \langle 3, q[i] = \mathsf{idle} \wedge x = 1 \rangle$. The only enabled transition is $\mathtt{t}(\mathsf{idle}, \mathsf{start})$, since $(q[i] = \mathsf{idle} \wedge x = 1) \Rightarrow (q[i] = \mathsf{idle})$ is satisfiable. For the

[3] We assume that there are no universal guards.

MUX-SEM example with $\mathsf{t} = \mathsf{t}(\mathsf{idle}, \mathsf{start})$, line 3 is:

$$\exists\, q[i] \in \mathsf{L}, x \in \mathbb{B} : (q[i] = \mathsf{idle} \wedge x = 1) \wedge (q[i] = \mathsf{idle}) \wedge (q[i]' = \mathsf{start} \wedge x' = x)$$
$$\equiv\ q[i]' = \mathsf{start} \wedge x' = 1.$$

Unpriming (line 5) yields the class formula $q[i] = \mathsf{start} \wedge x = 1$ with a count of 1, yielding the anonymized class $\langle 1, q[i] = \mathsf{start} \wedge x = 1 \rangle$. We assume $post(\mathsf{t}, \mathtt{C},)$ yields a class with count one (line 10), and we handle combining classes with equal formulas shortly.[4]

The new anonymized state representation $\mathtt{S}_{\mathtt{New}}$ is constructed using the classes of the anonymized state representation $\mathtt{S}$ of the current iteration along with the new anonymized class, $\mathtt{C}_{\mathtt{New}}$ (lines 12 through 19). First, the classes for $\mathtt{S}_{\mathtt{New}}$ are set to be the anonymized classes of $\mathtt{S}$, without the anonymized class of the current iteration, $\mathtt{C}$ (line 12) For MUX-SEM, this yields:

$$\mathtt{S}_{\mathtt{New}}.\mathtt{Classes} = \{\},$$

since the only element in $\mathtt{S}.\mathtt{Classes}$ is $\langle 3, q[i] = \mathsf{idle} \rangle$.

Next, if the class count of $\mathtt{C}$ is not one, then it is added to the classes of $\mathtt{S}_{\mathtt{New}}$, with its count reduced by to indicate some automaton has left the set of states satisfying the corresponding class formula (line 14). On the other hand, if $\mathtt{Count} = 1$, that means the old anonymized class $\mathtt{C}$ would no longer have any automata with states satisfying its class formula, so it is not added to the new anonymized state representation's classes. For MUX-SEM, this yields:

$$\mathtt{S}_{\mathtt{New}}.\mathtt{Classes} = \{\langle 2, q[i] = \mathsf{idle} \rangle\}.$$

Finally, the new anonymized class $\mathtt{C}_{\mathtt{New}}$ is added to the classes of $\mathtt{S}_{\mathtt{New}}$ (line 18). For MUX-SEM, this yields:

$$\mathtt{S}_{\mathtt{New}}.\mathtt{Classes} = \{\langle 2, q[i] = \mathsf{idle} \rangle, \langle 1, q[i] = \mathsf{start} \rangle\}.$$

[4] An alternative to combining classes with equivalent formulas is to find any classes in the anonymized state representation with equivalent formulas to the one just computed and increment their counts.

```
1  function mergeClasses(S) {
     foreach anonymized class C1 in S.Classes {
3      foreach anonymized class C2 in S.Classes {
         // only check equivalence of distinct (pointer-wise) elements in S.Classes
5        if C1 = C2 {
           continue;
7        }
         // attempt to prove classes have equivalent class formulas
9        if C1.Form ≢ C2.Form is UNSAT {
           // if so, merge their counts
11         C1.Count ← C1.Count + C2.Count
           S.Classes ← S.Classes \ {C2}
13       }
       }
15   }
     return S
17 }
```

Figure 4.6: The mergeClasses function combines any classes with equivalent class formulas and sums their counts. The input is a newly computed anonymized state S.

However, this may result in two classes with equal formulas, since the algorithm has not yet detected if any other classes had the same formula and assumed the new class $\texttt{C}_{\texttt{New}}$ had a count of one. Next, any other classes with the same formulas in $\texttt{S}_{\texttt{New}}$ are merged using mergeClasses (Figure 4.6 called from line 19).

Merging Classes. The function mergeClasses (Figure 4.6) takes an anonymized state representation S and returns another anonymized state representation that is guaranteed to not have any classes with equal formulas. Thus, mergeClasses finds all formulas in the classes of S.Classes that are equal and takes the sum of their counts for one class (line 11) and removes the other (line 12). Proving two class formulas $\texttt{Form}_1$ and $\texttt{Form}_2$ are equal is accomplished by checking that $\neg(\texttt{Form}_1 = \texttt{Form}_2)$ is unsatisfiable.

After merging any classes with equivalent formulas, mergeClasses returns the (potentially modified) anonymized state representation (line 16). This returns the anonymized state representation back to *post*, which then returns the (potentially) new anonymized state representation to discreteSuccessors (Figure 4.5, line 20). Finally, if the new anonymized state representation is actually new—in that there are no equivalent anonymized state representations already contained in the reachable states, newly computed states, or frontier—then it is added to the frontier (Figure 4.4, line 7). The loops of discreteSuccessors then continue, eventually computing all the discrete successors, which is returned as a set of anonymized

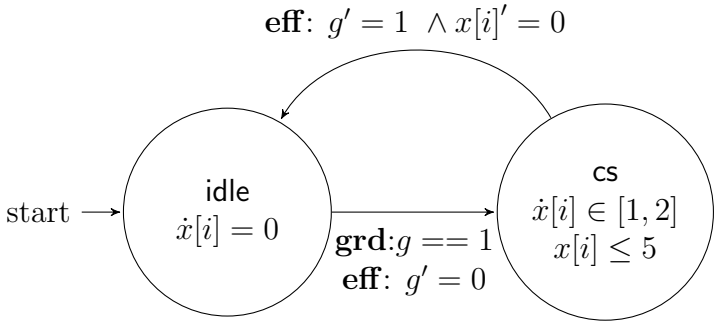

Figure 4.7: MUX-SEM-RA mutual exclusion algorithm for automaton $\mathcal{A}(i)$ for illustrating the computation of continuous successors in the anonymized state-space representation.

state representations to symreach line 13. This ends the computation of discrete successors, so the next step of symreach is computing continuous successors.

4.3.2 Continuous Successors

We will utilize a mutual exclusion algorithm, MUX-SEM-RA, shown in Figure 4.7 for illustrating the computation of continuous successors. The continuous successors are computed using the function contSuccessors—shown in Figure 4.8—called from symreach (Figure 4.3, line 13). The MUX-SEM-RA algorithm has two locations, idle and cs, one continuous local variable $x[i]$, and a global Boolean semaphore g. The continuous variable $x[i]$ increases at a rate between 1 and 2 while in location cs, up to a maximum of 5.

For an anonymized state representation S, the continuous successors contSuccessors computes an overapproximation of the post-states reachable by trajectories from the anonymized state S. The function contSuccessors takes three arguments: an anonymized state representation S, the current reachable states AnonReach, and the newly computed frontier $\mathsf{Frontier}_{\mathtt{New}}$. The output of contSuccessors is a set of anonymized states, $\mathsf{States}_{\mathtt{New}}$, which represent the updates to any continuous variables according to the semantics of trajectories (recall Section 2.4.3). We assume that the hybrid automaton template $\mathcal{A}(i)$ used to construct the network $\mathcal{A}^{\mathsf{N}}$ does not contain any stopping conditions. The next step is to call the function $post_{time}$ with an anonymized state representation S from the frontier (Figure 4.9 called at line 3).

For an anonymized state S in the frontier, $post_{time}$ computes an overapproximation of the

```
1  function contSuccessors(S, AnonReach, Frontier_New) {
     States_New ← ∅
3    S_New ← post_time(S) // compute continuous successors from S
     S_New ← mergeClasses(S_New)
5    if S_New ∉ (AnonReach ∪ States_New ∪ Frontier_New) {
       States_New ← States_New ∪ {S_New}
7    }
     return States_New
9  }
```

Figure 4.8: Computation of the continuous successors from an anonymized state $\mathtt{S}$. The inputs are an anonymized state $\mathtt{S}$ from the frontier, the reachable states AnonReach, and the newly computed frontier Frontier from the current iteration of the breadth-first reachability computation.

post-states from $\mathtt{S}$ owing to the individual trajectories of all automata in the network for up to the most amount of time that can elapse before any invariant is violated. The anonymized state specifies a location $\mathsf{m} \in \mathsf{L}$ for each automaton in the network (recall Assumption 4.4), and each location m specifies a trajectory statement, so trajectories are defined for each automaton in the network. For the following explanation, suppose the anonymized state of MUX-SEM-RA from which the continuous successors being computed is:

$$\mathtt{S} \;=\; \langle \{ \langle 2, 1, q[i] = \mathsf{idle} \;\wedge\; x[i] = 0 \rangle, \langle 1, 1, q[j] = \mathsf{cs} \;\wedge\; x[j] = 0 \rangle \}, g = 0 \rangle\,, \tag{4.15}$$

that is, there are two classes of automata, where the first class has two automata with locations idle and $x[i]$ equal to zero, the second class has one automaton with location cs and $x[j]$ equal to zero, and the global semaphore is 0.

Each new anonymized state $\mathtt{S}_{\mathtt{New}} \in \mathsf{States}_{\mathtt{New}}$ computed corresponds to the trajectory semantics updating the continuous variables of *all* automata in the network $\mathcal{A}^{\mathsf{N}}$. We use the variable $postFormula$ as a formula encoding the trajectory semantics of all automata in the network $\mathcal{A}^{\mathsf{N}}$ (line 3), which is initially the constraint $t_e > 0$, indicating that some positive real amount of time t_e will elapse. However, for an anonymized state $\mathtt{S}$, for distinct anonymized classes $\mathtt{C}_1$, $\mathtt{C}_2$ in $\mathtt{S.Classes}$, the symbolic indices appearing in the formulas may be equal, $\exists i \in ivars(\mathtt{C}_1)$ and $\exists j \in ivars(\mathtt{C}_2)$ such that $i = j$. Since $postFormula$ will encode the states of all automata in the network, the symbolic index variables appearing in any class formula of any anonymized class must be distinct. Rather than performing these tedious syntactic manipulations, we assume that for an anonymized state $\mathtt{S}$, for distinct classes $\mathtt{C}_1$,

```
1  function post_time(S) {
     // formula used to encode semantics of trajectories for all automata in the network
3    postFormula ← t_e > 0
     // set of bound variables used in computing the successors
5    bound ← ∅
     // iterate over each class in the symmetric state
7    foreach anonymized class C in S.Classes {
       // encode the prestate condition enforced by the class formula
9      postFormula ← postFormula ∧ C.Form
       // determine the locations any automata may be in each class (recall Assumption 4.4)
11     foreach location m in L {
         // iterate over all indices (ranks)
13       foreach i in {i_1,...,i_C.I} {
           // use location m if automaton i is in m
15         if C.Form ⇏ (q[i] = m) is UNSAT {
             // add the trajectory semantics overapproximating the post-states
17           postFormula ← postFormula ∧ inv(m, i) ∧ X[i] ∈ flow(m, X[i], t_e)
             bound ← bound ∪ V_i
19         }
         }
21     }
     }
23   bound ← bound ∪ {t_e}
     postFormula ← QuantElim(∃ bound : postFormula)
25   postFormula ← Substitute(postFormula, V'_i, V_i)
     S_New ← RemapClasses(S, postFormula)
27   return S_New
   }
```

Figure 4.9: $post_{time}$ function that computes the continuous successors from an anonymized state S.

$\mathtt{C}_2$ in S.Classes, $\forall i \in ivars(\mathtt{C}_1)$, $\forall j \in ivars(\mathtt{C}_2)$, we have $i \neq j$.[5] The variable *bound* (line 5) is a set of variable names used for computing the continuous successors using quantifier elimination.

First, we iterate over each class C in the prestate anonymized representation S's classes, S.Classes (line 7). The constraints on the prestates are specified by each class formula, which are conjuncted with *postFormula* (line 9). Each anonymized class formula C.Form of an anonymized state representation S specifies the location(s) the automata are in (recall Assumption 4.4), so the first step is to determine the dynamics that will modify each class formula. This is accomplished by first determining the appropriate flow-rate conditions to use for each class in S.Classes, which can be detected by finding which Form imply the location variable $q[i]$ is in some location $\mathtt{m} \in \mathtt{L}$. If the control location of automaton i is found to be equal to location m, then the trajectory statement of location m is used to define the semantics of the time-evolution of i's continuous variables (line 17). The control location

[5] This is a tedious, but trivial invariant that we maintain in our implementation, so we make this assumption for clarity of presentation only.

determination is performed by first iterating over the locations (line 11), and next iterating over all the index variables in the class formula (line 13). The determination of the control location of each class is accomplished by the check of line 15.

If any process in any class is in control location m, then a formula encoding the trajectory semantics of location m are conjuncted to the formula *postFormula* (line 17). The semantics of trajectories result in *all* the automata's continuous variables evolving over time t_e, so the formula encoding the trajectory statements of all automata is conjuncted (line 17). At this point, *postFormula* contains the constraints on variables specified by the class formula, the global variables, flow conditions, and any invariants defining the trajectory semantics (recall Section 2.4.3). Finally, the unprimed variables are added to the set of bound variables to be eliminated (line 18). This process continues for all anonymized classes in S and all indices. For the MUX-SEM-RA example with anonymized state from Equation 4.15, this process yields:

$$\begin{aligned} \mathit{postFormula} = {} & q[i] = \mathsf{idle} \ \wedge x[i] = 0 \ q[i]' = q[i] \ \wedge \ x[i]' = x[i] + 0 * t_e \ \wedge \ q[j] = \mathsf{cs} \ \wedge \\ & x[j]' \geq x[j] + 1 * t_e \ \wedge \ x[j]' \leq x[j] + 2 * t_e \ \wedge \ x[j]' \leq 5 \ \wedge q[j]' = q[j] \ \wedge \\ & g = 0 \ \wedge \ g' = g \ \wedge \ t_e > 0, \text{ and} \\ \mathit{bound} = {} & \{q[i], x[i], q[j], x[j], g\}. \end{aligned} \tag{4.16}$$

Once all classes and all corresponding indices have had trajectory statements added to *postFormula*, the elapsing time variable t_e is added to the bound variables (line 23). The post-states are computed by existentially quantifying and eliminating the variables in the set *bound* (line 24) and then renaming primed variables with their unprimed counterparts (line 25).[6] For the MUX-SEM-RA example with *postFormula* and *bound* from Equation 4.16, this yields the formula:

$$\mathit{postFormula} = \ q[i] = \mathsf{idle} \ \wedge \ x[i] = 0 \ \wedge \ q[j] = \mathsf{cs} \ \wedge \ 0 < x[j] \leq 5 \ \wedge \ g = 0. \tag{4.17}$$

[6]This may result in a DNF formula. If this is the case, each conjunctive clause is created as a new anonymized state representation by iterating over the conjunctive clauses so that all class formulas are CNF formulas.

```
    function RemapClasses(S, postFormula) {
 2    S_New.Classes = ∅
      // project postFormula onto the variables of indices in each class in S
 4    foreach anonymized class C_1 in S.Classes {
        bound ← ∅
 6      foreach anonymized class C_2 in S.Classes {
          if C_1 ≠ C_2 {
 8          foreach i in {i_1,...,i_C2.I} {
              bound ← V_i
10          }
          }
12      }
        // postTmp will be a formula only over the variables of indices of class C_1
14      postTmp ← QuantElim(∃ bound : postFormula)
        // create new class with formula corresponding to post-states
16      C_New.Form = postTmp
        // copy the prestate class count of the corresponding class
18      C_New.Count = C_1.Count
        // add new class to the new anonymized state's classes
20      S_New.Classes ← S_New.Classes ∪ C_New
      }
22    S_New.N ← S.N
      return S_New
24  }
```

Figure 4.10: Function for remapping variables in *postFormula* to their prestate index names and class counts to create the new anonymized state representation $\mathtt{S}_{\mathtt{New}}$. The function first projects onto the variables with index names of each class in the prestate and then uses the prestate count to ensure class counts remain constant over trajectories.

Finally, we call the function RemapClasses with S—the prestate anonymized representation—and *postFormula*—the formula encoding the post-state constraints—to remap subformulas of *postFormula* to their prestate classes (Figure 4.10 called at line 26). This is done to ensure the class counts are constant when computing post-states due to trajectories.

The function RemapClasses recreates the classes from the prestate S from which the continuous successors are being computed. That is, it determines the appropriate class counts for each new subformula in *postFormula* corresponding to an anonymized class in the prestate S. Essentially, RemapClasses determines which indices appearing in *postFormula* correspond to which classes in the prestate S, so that the appropriate class counts can be determined, and accomplishes this in two steps. Since we assumed all classes contain distinct index variables, this is done by iterating over each class C in the prestate S.Classes, then projecting onto the C.I symbolic indices for C and using the count C.Count.

First, we iterate over all classes in the prestate anonymized state S (line 4). Next, we project away all the variables of automata with indices coming from different classes (i.e., other than $\mathtt{C}_1$) from *postFormula* (lines 6 through 14). This is done by collecting all the

variable names corresponding to other classes (line 6), then projecting them away using quantifier elimination, yielding a formula $postTmp$ (line 14).

Next, the new anonymized class $\mathtt{C}_{\mathtt{New}}$ is created using the $postTmp$ formula and the class count of the prestate class $\mathtt{C}_1$ (line 18). Lastly, $\mathtt{C}_{\mathtt{New}}$ is added to the set of anonymized classes for the new anonymized state representation $\mathtt{S}_{\mathtt{New}}$. This process repeats for all the classes in the prestate anonymized state representation $\mathtt{S}$ from which the continuous successors are being computed. Once this has been done for all classes in $\mathtt{S.Classes}$, the new anonymized state representation $\mathtt{S}_{\mathtt{New}}$ is returned by RemapClasses (line 23). For the MUX-SEM-RA example with $postFormula$ from Equation 4.17, this yields the new anonymized state:

$$\mathtt{S}_{\mathtt{New}} \;=\; \langle \{ \langle 2, 1, q[i] = \mathsf{idle} \;\wedge\; x[i] = 0 \rangle , \langle 1, 1, q[j] = \mathsf{cs} \;\wedge\; 0 \leq x[j] \leq 5 \rangle \} , g = 0 \rangle .$$

The new anonymized state representation $\mathtt{S}_{\mathtt{New}}$ is subsequently returned by $post_{time}$ (Figure 4.9, line 27). Finally, as in the discrete successors case, if the new anonymized state representation $\mathtt{S}_{\mathtt{New}}$ is actually new—in that there are no equivalent anonymized state representations already contained in the reachable states, newly computed states, or frontier—then it is added to the frontier (Figure 4.8, line 5).

The process of computing discrete and continuous successors then continues for any remaining anonymized state representations in Frontier, then repeats for any states added to the new frontier $\mathsf{Frontier}_{\mathtt{New}}$.

4.4 Analysis of Reachability Algorithm Using Anonymized States

This section presents analysis of properties for the reachability algorithm using anonymized states (Figure 4.3).

The next invariant of the algorithm in Figure 4.3 states that no two class formulas in a reachable anonymized state representation $\mathtt{S}$ are equal, and it follows from the definition of mergeClasses (Figure 4.6), which merges any two classes with equal formulas prior to them being added to AnonReach.

Invariant 4.5 *For any* $\mathtt{S} \in \mathsf{AnonReach}$*, for every* $\mathtt{C}_1, \mathtt{C}_2 \in \mathtt{S.Classes}$*,* $\mathtt{C}_1.\mathtt{Form} \not\equiv \mathtt{C}_2.\mathtt{Form}$*.*

The next invariant states there is never an anonymized state in the new frontier $\mathsf{Frontier}_{\texttt{New}}$ that is already in the anonymized reachable states AnonReach. This is maintained by both the computations of the discrete successors discreteSuccessors and contSuccessors since only anonymized state representations that are not equivalent to any anonymized state representation in the union of AnonReach, Frontier, and the new anonymized state representations $\mathsf{States}_{\texttt{New}}$ are added to $\mathsf{Frontier}_{\texttt{New}}$ (Figure 4.4, line 7 and Figure 4.8, line 5).

Invariant 4.6 *For any* $\mathtt{S}_1 \in \mathsf{Frontier}_{\texttt{New}}$*, for any* $\mathtt{S}_2 \in \mathsf{AnonReach}$*,* $\mathtt{S}_1 \not\equiv \mathtt{S}_2$.

The next invariant states that no two anonymized state representations in the anonymized reachable states are equivalent.[7] It follows immediately from Invariant 4.6.

Invariant 4.7 *For any distinct* $\mathtt{S}_1, \mathtt{S}_2 \in \mathsf{AnonReach}$*,* $\mathtt{S}_1 \not\equiv \mathtt{S}_2$.

The next invariant states that the sum of all the class counts `Count` equals N. It follows from the definitions of discreteSuccessors and contSuccessors, since discreteSuccessors always decreases class counts by the same amount it increases them—so the sum remains invariant—and contSuccessors does not change class counts, only class formulas. Additionally, mergeClasses changes class counts, but their sum remains the same since it removes any duplicate classes after adding their counts (Figure 4.6, lines 11 through 12).

Invariant 4.8 *For any* $\mathtt{S} \in \mathsf{AnonReach}$*,*

$$\mathsf{N} = \sum_{\mathtt{C} \in \mathtt{S.Classes}} \mathtt{C.Count}.$$

Theorem 4.9 states partial correctness of the reachability algorithm, namely soundness, which is that the concretization of the set of anonymized reachable states AnonReach contains the actual set of reachable states for the particular finite instantiation. The theorem follows since both the continuous and discrete successors are computed as overapproximations, as described when defining discreteSuccessors and contSuccessors.

[7] Of course, if we view AnonReach as a set, then this is by definition, however, AnonReach is a data-structure in the algorithm, so this is an invariant the algorithm must maintain.

```
parameter name='lb' type='real' value = 1.0 // minimum rate
parameter name='ub' type='real' value = 2.0 // maximum rate
parameter name='B' type='real' value = 5.0  // guard constant

automaton name='MUX-INDEX-RECT'
  variable name='q[i]' type='L'     // location local variable
  variable name='x[i]' type='real' // continuous local variable
  variable name='g' type='index'   // global lock variable

  location name='rem'
    flowrate: x[i]_dot >= lb and x[i]_dot <= ub
  location name='try'
    flowrate: x[i]_dot >= lb and x[i]_dot <= ub
  location name='cs'
    flowrate: x[i]_dot >= lb and x[i]_dot <= ub

  transition from='rem' to='try'
    grd: g = ⊥ and x[i] >= B
    eff: g′ = i and x[i]′ = 0.0
  transition from='try' to='cs'
    grd: g = i and x[i] >= 2*B
    eff: x[i]′ = 0
  transition from='cs' to='rem'
    grd: x[i] >= 3*B
    eff: x[i]′ = 0
property: forall i, j (i != j and q[i] = cs) implies (q[j] != cs)
initially: forall i (q[i] = rem and x[i] = 0 and g = ⊥)
```

Figure 4.11: *Passel* input file specifying automaton template $\mathcal{A}(i)$ for mutual exclusion algorithm MUX-INDEX-RECT.

Theorem 4.9 *For a fixed* $\mathsf{N} \in \mathbb{N}$*, for the network* $\mathcal{A}^{\mathsf{N}}$ *composed of* N *instantiations of the hybrid automaton template* $\mathcal{A}(\mathsf{N}, i)$*, the anonymized reachable states* AnonReach *computed by the algorithm in Figure 4.3 are an overapproximation of the reachable states of* $\mathcal{A}^{\mathsf{N}}$*:*

$$\mathsf{Reach}(\mathcal{A}^{\mathsf{N}}) \subseteq [\![\mathsf{AnonReach}]\!] .$$

The approximation comes from two sources. First, index-typed variables are abstracted to be equal or not equal to some index only. Second, the rectangular dynamics are overapproximated. In the case that no index-typed variables are included, the computation is exact, so $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}}) = [\![\mathsf{AnonReach}]\!]$.

4.4.1 Example with Anonymized Reachable State-Space Cardinality Independent of N

This section describes a simple timed mutual exclusion algorithm that has an anonymized reach set that is *independent of* N. The hybrid automaton template $\mathcal{A}(\mathcal{N}, i)$ specifying the

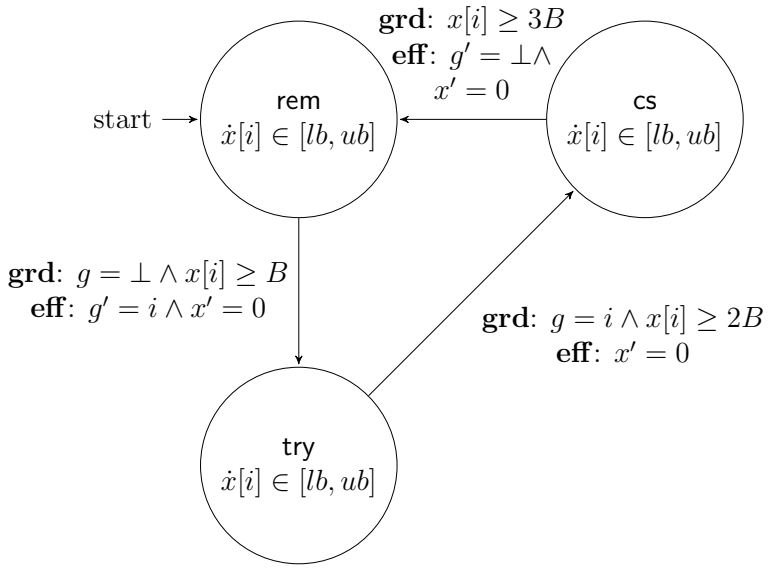

Figure 4.12: MUX-INDEX-RECT mutual exclusion algorithm of Figure 4.11.

MUX-INDEX-RECT example appears in Figure 4.11 and graphically in Figure 4.12. Specifically, the number of anonymized classes in `ReachForms` does not increase as a function of N. Additionally, for any S ∈ AnonReach, the sum of the class counts of S is 1, N, or N − 1. This is in contrast to the MUX-SEM example described previously, which has some S with counts in $\{1, \ldots, \mathsf{N}\}$, so their runtimes and memory usages increase as a function of N. Due to this state-space size independence from N, our experiments have been successful for computing the reachable states for compositions of millions of automata.

4.5 Summary

In this chapter, we present an on-the-fly forward reachability algorithm that computes an anonymized representation of the reachable states for finite instantiations of parameterized networks of hybrid automata. The anonymized representation avoids generating all permutations of automata indices and states, and in Section 7.6, we show it to be effective at computing the reachable states of networks with hundreds of automata for some special examples.

Chapter 5

Proving Inductive Invariants

In this chapter, we present a uniform verification method for safety properties of parameterized networks of hybrid automata of the type defined in Chapter 2, Definition 2.2. Each automaton is equipped with a finite collection of pointers to other automata that enables it to read their state. This chapter presents a small model result for such networks that reduces the verification problem for a system with arbitrarily many processes to a system with finitely many processes. The result is applied to verify and discover counterexamples of inductive invariant properties for distributed protocols like Fischer's mutual exclusion algorithm and the Small Aircraft Transportation System (SATS). The method is implemented to check inductive invariants automatically in the *Passel* verification tool (see Chapter 7). *Passel* automatically proves safety properties for parameterized networks of hybrid automata by using a combination of invariant synthesis and inductive invariant proving. This chapter describes the inductive invariance checks, and synthesizing candidate invariants is described in Chapter 6.[1]

5.1 Introduction

This chapter presents a method for uniform verification of parameterized networks of hybrid automata, that is, networks composed of arbitrarily many instantiations of some template automaton. For uniform verification, a property $\zeta(\mathcal{N})$ and a hybrid automaton template $\mathcal{A}(\mathcal{N}, i)$—written $\mathcal{A}(i)$ in short—are given. The property must be independent of the number and the identities of the modules, and we must verify that $\zeta(\mathcal{N})$ holds for any system built

[1] This chapter is based in part on prior work [3], portions of which are reprinted here with permission.

from arbitrarily many instances of $\mathcal{A}(i)$. That is,

$$\forall\, \mathcal{N} \in \mathbb{N},\ \mathcal{A}_1 \| \mathcal{A}_2 \| \ldots \| \mathcal{A}_{\mathcal{N}} \models \zeta(\mathcal{N}), \tag{5.1}$$

where the precise meaning of the parallel composition of operator $\|$ is as given in Definition 2.2 in Chapter 2 of this dissertation.

To perform verification of an arbitrary number of hybrid automata, we develop and use a small model result for the subclass of hybrid automaton networks with rectangular dynamics. In formal logic, small model theorems are used to prove decidability of satisfiability checking for formulas of different logical classes. Many prefix classes of FOL were shown to be decidable by showing that the class has the *finite model property*: every satisfiable formula also has a finite model [159]. They have been applied to verification of concurrent and multi-threaded programs [41–43, 96–98, 160]. The idea behind applying them in verification is to identify classes of systems and specifications with the small model property, which reduces an infinite problem to a finite one. In many cases, the proof of the finite model property comes with an explicit bound on the size of the finite structure that may satisfy (or violate) the property in question. For uniform verification, small model theorems provide a finite threshold N_0 such that if, for all $\mathsf{N} \leq \mathsf{N}_0$, $\mathcal{A}^{\mathsf{N}} \triangleq \mathcal{A}_1 \| \ldots \| \mathcal{A}_{\mathsf{N}} \models \zeta(\mathsf{N})$, then Equation 5.1 also holds. In the context of verification of discrete parameterized networks, small model theorems were developed in [41, 42] and also studied in [43, 45, 46], and have only previously been studied in the context of purely discrete system to the best of our knowledge.

The contributions of this chapter are:

(a) A small model theorem for hybrid automata networks that guarantees the existence of a bound N_0, such that, if an instantiation of $\mathcal{A}^{\mathcal{N}}$ violating $\zeta(\mathcal{N})$ exists for some $\mathcal{N} > \mathsf{N}_0$, then $\mathcal{A}^{\mathsf{N}_0}$ must also violate $\zeta(\mathsf{N}_0)$. Thus, the verification problem from Equation 5.1 is solved if, for all $\mathsf{N} \leq \mathsf{N}_0$, no instantiation $\mathcal{A}^{\mathsf{N}}$ violates $\zeta(\mathsf{N})$.

(b) The theorem is applied in the *Passel* verification tool that we use to automatically check inductive invariants up to the bound N_0 by querying the satisfiability modulo theories (SMT) solver Z3 [136].

(c) To the best of our knowledge, this is the first result on automatic uniform verification of parameterized networks of hybrid automata, although there are other techniques for uniformly verifying networks of timed automata [32, 33, 47, 49, 50].

The input to *Passel* is a hybrid automaton specification $\mathcal{A}(i)$ and a candidate safety property $\zeta(\mathcal{N})$. Then, the bound N_0 is computed from the syntactic description of $\mathcal{A}(i)$ and $\zeta(\mathcal{N})$. In addition to these hybrid protocols, we have also verified several purely discrete algorithms—cache coherence protocols and mutual exclusion algorithms like the simplified bakery algorithm— studied in [51, 76]. The experiments (see Section 7.7) indicate that it is feasible to develop automatic methods relying on our small model result that apply both to distributed cyber-physical systems and classic distributed algorithms. The success of our experiments in part relies on the strengths of state-of-the-art SMT solvers like Z3, which allow for quantified formulas and have quantifier elimination and instantiation procedures for real and integer arithmetic [161, 162]. One weakness of the method is that it will fail unless an inductive invariant is given. The user must provide candidate inductive invariants that are strong enough to imply the desired invariant property. Alternatively, methods can be used to attempt to find such candidates automatically, and we present some in Chapter 6.

Outline. In Section 5.1.1, we outline the general methodology of proving inductive invariants for parameterized networks of hybrid automata, which requires a set of candidate inductive invariants to either be provided by a user or synthesized in some way. In Section 5.2, we present the syntax for a restricted class of *Passel* assertions—introduced earlier in Section 2.3.2—which we call LH-assertions. When LH-assertions are used for specifying various syntactic components of the hybrid automaton, we show a small model theorem in Section 5.2. We then show in Section 5.3 how inductive invariant properties for parameterized networks of hybrid automata can be asserted in terms of LH-assertions, and we conclude in Section 5.4.

```
1  function inductiveInvariance(𝒜(𝒩,i), ζ(𝒩), Init, N, P) {
     // synthesize candidate inductive invariants from finite instances
3    ψ(i_1,...,i_P) ← synthesis(𝒜(N,i), Init, N, P)
     Γ(𝒩) ← ∀i_1,...,i_P ∈ [𝒩] : ψ(i_1,...,i_P)
5
     // inductive invariance check for any 𝒩
7    if (∀𝒩 ∈ ℕ Init(𝒩) ⇒ Γ(𝒩) is valid and // per Definition 2.4 (A)
         ∀𝒩 ∈ ℕ transitionConsecution(𝒜(𝒩,i), 𝒩, Γ(𝒩)) is valid and // per Definition 2.4 (B)
9        ∀𝒩 ∈ ℕ trajectoryConsecution(𝒜(𝒩,i), 𝒩, Γ(𝒩)) is valid and // per Definition 2.4 (C)
         ∀𝒩 ∈ ℕ Γ(𝒩) ⇒ ζ(𝒩) is valid) { // per Definition 2.8
11         return ζ(𝒩) is invariant for all 𝒩
     }
13   else {
       return potential counterexample
15   }
   }
```

Figure 5.1: Inductive invariance proof method with auxiliary invariant synthesis. The inputs are an automaton specification $\mathcal{A}(\mathcal{N}, i)$, a desired safety property $\zeta(\mathcal{N})$, an initial condition assert Init, and two constants, N and P. The output is either a proof of the safety property $\zeta(\mathcal{N})$ for all $\mathcal{N} \in \mathbb{N}$, or a potential counterexample. The latter either indicates $\mathcal{A}(\mathcal{N}, i)$ has a bug and does not satisfy $\zeta(\mathcal{N})$ or that the synthesized invariants are not strong enough to prove $\zeta(\mathcal{N})$.

5.1.1 Proving Inductive Invariants

Passel attempts to automatically verify safety properties of $\mathcal{A}^{\mathcal{N}}$ that hold for any $\mathcal{N} \in \mathbb{N}$ by checking and synthesizing inductive invariants. This chapter describes methods for checking if an assertion $\Gamma(\mathcal{N})$ is an inductive invariant for any $\mathcal{N}$ (Definition 2.4). For this chapter, we assume such candidate assertions are given, but Chapter 6 presents methods for finding (synthesizing) candidate inductive invariants, which can then be checked using the methods in this chapter. The overall methodology of finding and checking inductive invariants is described by the pseudocode of Figure 5.1.

5.2 Small Model Theorem

In this section, we present the main small model result (Theorem 5.2) for the restricted class of *Passel* assertions called LH-assertions. Thus, for a specific inductive invariant $\Gamma(\mathcal{N})$, Theorem 5.2 provides a threshold on size of models, written N_0. If for all $\mathsf{N} \leq \mathsf{N}_0$, $\Gamma(\mathsf{N})$ is an inductive invariant for $\mathcal{A}^{\mathsf{N}}$, then $\Gamma(\mathsf{N})$ is indeed an inductive invariant for all $\mathcal{N} \in \mathbb{N}$. This makes it possible to verify inductive invariants for parameterized networks of hybrid

automata $\mathcal{A}^{\mathcal{N}}$ by verifying $\Gamma(\mathsf{N})$ for a finite number of instances of $\mathcal{A}^{\mathsf{N}}$. If the syntactic components of the hybrid automaton template $\mathcal{A}(\mathcal{N}, i)$ are specified with this restricted class, then the small model theorem can be applied (Theorem 5.2).

Definition 5.1 *An* LH-assertion *is an index sentence—refer to Section 2.3.2—of the form*

$$\forall t_1 \in \mathbb{R}\ \forall i_1, \ldots, i_k \in [\mathcal{N}]\ \exists t_2 \in \mathbb{R}\ \exists j_1, \ldots, j_m \in [\mathcal{N}]\ :\ \varphi(t_1, i_1, \ldots, i_k, t_2, j_1, \ldots, j_m),$$

where φ is a quantifier-free Passel assertion containing no unbounded integer variables.

It is essential that the order (shape) of quantifiers is as specified ($\forall^*\exists^*$) for establishing small model theorems like Theorem 5.2. We mention that t_e and t_p are only used to respectively model an elapse of time of length t_e and enforcing invariants for all trajectories of lengths $0 \leq t_p \leq t_e$ (refer to the semantics of continuous trajectories defined in Section 2.4.2).

We provide several example LH-assertions:

$$\forall i, j : i \neq j \implies (q[j] = q[j] \implies |x[j] - x[i]| > a), \tag{5.2}$$

$$\forall i, j : i = j \ \vee\ q[i] = q[j] \ \vee\ (x[j] - x[i] - a < 0) \ \vee\ (x[i] - x[j] - a < 0), \text{ and} \tag{5.3}$$

$$\forall i\ \exists j : p[i] = j \wedge |x[i] - x[p[i]]| > a. \tag{5.4}$$

We reiterate that we only use LH-assertions with t_e and t_p for checking the inductive invariance conditions for continuous trajectories as shown in Section 5.3. Reading these assertions as statements about networks of automata, the first one states that all automata with identical values of the discrete local variable $q[i]$ have a minimum gap of a between the values of their $x[i]$ variables. The first assertion is an abbreviation of the second. The last assertion states every automaton has a pointer $p[i]$ to another automaton and that there is a minimum separation of a between its $x[i]$ value and the $x[p[i]]$ value of the automaton to which $p[i]$ points.

Theorem 5.2 *Let $\Gamma(\mathcal{N})$ be an LH-assertion of the form $\forall t_e \in \mathbb{R}\ \forall i_1, \ldots, i_k \in [\mathcal{N}]\ \exists t_p \in \mathbb{R}$ $\exists j_1, \ldots, j_m \in [\mathcal{N}] : \varphi(t_e, i_1 \ldots, i_k, t_p, j_1, \ldots, j_m)$, where φ is a quantifier-free formula involving the index variables i_1, …, i_k, j_1, …, j_m, real variables t_e and t_p, and global and*

local variables in V_i*, where* $i \in ivars(\varphi)$*. Then,* $\forall \mathcal{N} \in \mathbb{N} : \Gamma(\mathcal{N})$ *is valid iff for all* $n \leq \mathsf{N}_0 = (e+1)(k+2)$*,* $\Gamma(\mathcal{N})$ *is satisfied by all* n*-models (recall Definition 2.3), where* e *is the number of index array variables in* φ *and* k *is the largest subscript of the universally quantified index variables in* $\Gamma(\mathcal{N})$*.*

Proof: If $\forall \mathcal{N} \in \mathbb{N} : \Gamma(\mathcal{N})$ is valid, then all its models satisfy it by the definition of validity.

For the other direction, we assume that all models of size n, for $n \leq (e+1)(k+2)$, satisfy $\Gamma(n)$. It suffices to show that $\psi \triangleq \forall \mathcal{N} \in \mathbb{N} : \Gamma(\mathcal{N})$ is valid. Suppose for the sake of contradiction that ψ is not valid. Then there exists a model M of size $n > (e+1)(k+2)$ that satisfies $\neg\psi \equiv \exists \mathcal{N}, t_e, i_1, \ldots, i_k : \forall t_p, j_1, \ldots, j_m : \neg\varphi$. We will show that for any model of size $n > (e+1)(k+2)$, there exists a model of size $n-1$ that contradicts the assumption that all n-models satisfy $\Gamma(n)$.

The n-model M assigns a real value to the variable t_e and values in $\{1, \ldots, n\}$ to the index variables i_1, ..., i_k (in addition to providing interpretations for the other variables and arrays). The values assigned to the universally quantified variables t_p, j_1, ..., j_m in the model M are not important, because any value of these variables would satisfy $\neg\psi$. The set of values assigned to $i_1, \ldots, i_k$ can contain at most k distinct values. Consider an index term with one of the forms 1, $\mathcal{N}$, or i_m, where i_m is an existentially quantified index variable in $\neg\psi$: any such term can take at most $k+2$ distinct index values. Thus, an index array term $p[i_m]$ can take at most $k+2$ distinct values. Since there are at most e index arrays, the set of all possible index arrays and terms can take at most $(e+1)(k+2)$ distinct values. Therefore, there exists a value in $\{1, \ldots, n\}$, say u, that is not assigned to any index variable or to any of the referenced values of the index arrays, in M.

Now, we define an $(n-1)$-model M' by removing u from $\{1, \ldots, n\}$ and shifting values assigned by M appropriately. The constant n is interpreted as $n-1$ in M'. For each index variable i_j, if $i_j < u$, then we assign $M'(i_j) = M(i_j)$, and otherwise we assign $M'(i_j) = M(i_j) - 1$, where the notation $M(v)$ is the assignment of v in model M. For each (index, discrete, or real) array $\bar{z}$, for each $i \in \{1, \ldots, n-1\}$, if $i < u$ then we assign $M'(z[i]) = M(z[i])$, and otherwise we assign $M'(z[i]) = M(z[i+1])$. Finally, it is routine to check that M' assigns the same binary value to each Atom in φ as M, and therefore M' also satisfies $\neg\psi$. This contradicts the assumption that all models of size n, for $n \leq (e+1)(k+2)$, satisfy

$\Gamma(n)$. ■

We close this section with the following result that lets us check the conditions of inductiveness over trajectories as assertions with the small model property. We model rectangular dynamics using an additional existential quantifier over reals in the time transition. The discussion that follows is how we are able to convert the relation **flow** used to define the set of continuous trajectories with a function $\mathbf{flow}_f$ defined below. An alternative would be to track upper and lower bounds of rectangular variables using two clocks, and convert to a timed automata as done in [30]. To define $\mathcal{T}^{\mathcal{N}}$ we first define the function $\mathbf{flow}(\mathbf{v}[i], \mathsf{m}, t)$ which returns a valuation $\mathbf{v}'[i]$, such that for each $v \in \mathsf{V}_i$ if v's type is not real, then $\mathbf{v}'[i].v = \mathbf{v}[i].v$, but otherwise, $\mathbf{v}'[i].v = \mathbf{v}[i].v + \mathbf{flowrate}(\mathsf{m}, v)t$.

Proposition 5.3 *Consider the flow function defined by* $\mathbf{flow}_f(\mathbf{v}[i], \mathsf{m}, t)$*, which returns a valuation* $\mathbf{v}'[i]$*, such that for each* $v \in \mathsf{V}_i$ *if* v*'s type is not real or its update type is not continuous, then* $\mathbf{v}'[i].v = \mathbf{v}[i].v$*, but otherwise,* $\mathbf{v}'[i].v = \mathbf{v}[i].v + \mathbf{flowrate}(\mathsf{m}, v)t$*, where* $\mathbf{flowrate}(\mathsf{m}, v)t = \delta t$*, for any* $\delta \in [a, b]$*. Alternatively, consider the flow relation defined by* $\mathbf{flow}_r(\mathbf{v}[i], \mathsf{m}, t)$*, which returns a set of valuations* $\mathcal{V}[i]$*, where for each* $\mathbf{v}'[i] \in \mathcal{V}[i]$*, such that for each* $v \in \mathsf{V}_i$ *if* v*'s type is not real or its update type is not continuous, then* $\mathbf{v}'[i].v = \mathbf{v}[i].v$*, but otherwise,* $\mathbf{v}[i].v + at \leq \mathbf{v}'[i].v \leq \mathbf{v}[i].v + bt$*.*

Recall from Section 2.4.3 that a pair $(\mathbf{v}, \mathbf{v}') \in \mathcal{T}^{\mathcal{N}}$ *iff:*

$$
\begin{aligned}
&\exists\ t_e \in \mathbb{R}_{\geq 0} :\ \forall\ i : [\mathsf{N}] : \exists\ \mathsf{l} \in \mathsf{Loc} : \\
&\quad \wedge\ \forall t_p \leq t_e : \mathbf{flow}(\mathbf{v}[i], \mathsf{l}, t_p) \models \mathbf{inv}(\mathsf{l}, i) \\
&\quad \wedge\ \forall t_p \leq t_e : \mathbf{flow}(\mathbf{v}[i], \mathsf{l}, t_p) \models \mathbf{stop}(\mathsf{l}, i) \Rightarrow t_p = t_e \\
&\quad \wedge\ \mathbf{v}'[i] \in \mathbf{flow}_r(\mathbf{v}[i], \mathsf{l}, t_e).
\end{aligned}
$$

Consider the alternative definition of $\mathcal{T}_f^{\mathcal{N}}$, where a pair $(\mathbf{v}, \mathbf{v}') \in \mathcal{T}_f^{\mathcal{N}}$ iff:

$$\begin{aligned}
&\exists\ t_e \in \mathbb{R}_{\geq 0} :\ \forall\ i : [\mathsf{N}] : \exists \delta \in \mathbb{R}_{\geq 0} \exists\ \mathsf{l} \in \mathsf{Mode}_i : \\
&\quad \wedge\, \forall t_p \leq t_e : \mathbf{flow}(\mathbf{v}[i], \mathsf{l}, t_p) \models \mathbf{inv}(\mathsf{l}, i) \\
&\quad \wedge\, \forall t_p \leq t_e : \mathbf{flow}(\mathbf{v}[i], \mathsf{l}, t_p) \models \mathbf{stop}(\mathsf{l}, i) \Rightarrow t_p = t_e \\
&\quad \wedge\, \mathbf{v}'[i] = \mathbf{flow}_f(\mathbf{v}[i], \mathsf{l}, t_e).
\end{aligned}$$

Then, the sets of trajectories under these definitions are the same, $\mathcal{T}^{\mathcal{N}} = \mathcal{T}_f^{\mathcal{N}}$.

Proof: We show $\mathcal{T}^{\mathcal{N}} \subseteq \mathcal{T}_f^{\mathcal{N}}$ and $\mathcal{T}_f^{\mathcal{N}} \subseteq \mathcal{T}^{\mathcal{N}}$. It is clear that $\mathcal{T}_f^{\mathcal{N}} \subseteq \mathcal{T}^{\mathcal{N}}$. For $\mathcal{T}^{\mathcal{N}} \subseteq \mathcal{T}_f^{\mathcal{N}}$, take any trajectory $\tau \in \mathcal{T}^{\mathcal{N}}$. The valuation of any variable v at state $\mathbf{v}'$ along τ satisfies $\mathbf{v}[i].v + at \leq \mathbf{v}'[i].v \leq \mathbf{v}[i].v + b[t]$. Consider a trajectory under the other semantics, where the first state along this trajectory $\mathbf{x}$ satisfies $\mathbf{x}[i].v = \mathbf{v}[i].v$ for each $i \in [\mathsf{N}]$ and each variable v. Suppose $\delta = \frac{1}{t}\int_0^t v(t)dt$, where $v(t)$ is the actual choice of $\mathbf{flowrate}(\mathsf{m}, \mathbf{v}[i].v)$ over the length of the trajectory. This integral must exist, and thus we have picked δ as the average flow rate over the trajectory of length t. Since $\mathbf{flowrate}(\mathsf{m}, \mathbf{v}[i].v) \in [a, b]$ for $a = \mathbf{lflowrate}$ and $b = \mathbf{uflowrate}$, which is a convex set, and $\delta \in [a, b]$ is also a convex set, we have that for this choice of δ, $\mathbf{x}[i].v + \delta t \in \tau$. Thus, $\tau \in \mathcal{T}_f^{\mathcal{N}}$. ■

5.3 Applying the Small Model Theorem to Check Inductive Invariants

For an automaton network $\mathcal{A}^{\mathcal{N}}$, an inductive invariant assertion is a logical sentence involving the variables in V_i (recall Definition 2.4). We require the invariant assertions to have all the universal quantifiers precede the existential quantifiers. Thus, an invariant assertion is of the form $\Gamma(\mathcal{N}) \triangleq \forall i_1, \ldots, i_k \in [\mathcal{N}] : \exists j_1, \ldots, j_m \in [\mathcal{N}] : \varphi$, where φ is a quantifier-free formula involving the index variables $i_1, \ldots, i_k$, $j_1, \ldots, j_m$, and the global and array variables in V_i for each $i \in \mathit{ivars}(\varphi)$.

For example, in the case of SATS, the assertion specifying a safe separation of aircraft is:

$$\forall i, j \in [\mathcal{N}] \ : \ (i \neq j \wedge q[i] = \mathsf{base} \wedge q[j] = \mathsf{base} \wedge next[j] = i) \Rightarrow x[i] - x[j] \geq L_S.$$

That is, if there is an aircraft i attempting to land, the aircraft immediately ahead of it is at least L_S distance away.

In the remainder of this section, we show how inductive invariant assertions for parameterized networks of hybrid automata can be stated as LH-assertions. For the purposes of this presentation, we assume that there are no global variables. It can be checked in a straightforward manner that these derivations hold for systems with global variables. We recall (Definition 2.4) that an assertion $\Gamma(\mathcal{N})$ is an inductive invariant for the parameterized network $\mathcal{A}^{\mathcal{N}}$ if, for all $\mathcal{N} \in \mathbb{N}$,

(A) **initiation**: for each state $\mathbf{v} \in \Theta^{\mathcal{N}} \Rightarrow \mathbf{v} \models \Gamma(\mathcal{N})$,

(B) **transition consecution**: for each $(\mathbf{v}, \mathbf{v}') \in \mathcal{D}^{\mathcal{N}}$, $\mathbf{v} \models \Gamma(\mathcal{N}) \Rightarrow \mathbf{v}' \models \Gamma(\mathcal{N})$, and

(C) **trajectory consecution**: for each $(\mathbf{v}, \mathbf{v}') \in \mathcal{T}^{\mathcal{N}}$, $\mathbf{v} \models \Gamma(\mathcal{N}) \Rightarrow \mathbf{v}' \models \Gamma(\mathcal{N})$.

We derive an LH-assertion for each of the above conditions.

From the definition of the initial states, $\mathbf{v} \in \Theta^{\mathcal{N}}$ iff $\forall i \in [\mathcal{N}] : \mathbf{v}[i] \models \mathsf{Init}_i$, where recall that Init_i is a formula involving the variables in V_i. Thus, condition (A) is equivalent to checking:

$$(\forall i \in [\mathcal{N}] : \mathsf{Init}_i) \Rightarrow (\forall i_1, \ldots, i_k \in [\mathcal{N}] : \exists j_1, \ldots, j_m \in [\mathcal{N}] : \varphi).$$

Moving the quantifiers of $\Gamma(\mathcal{N})$ to the front, we obtain:

$$\begin{aligned} &\forall i_1, \ldots, i_k \in [\mathcal{N}] : \exists j_1, \ldots, j_m \in [\mathcal{N}] : (\forall i \in [\mathcal{N}] : \mathsf{Init}_i \Rightarrow \varphi), \\ \equiv\ &\forall i_1, \ldots, i_k \in [\mathcal{N}] : \exists i, j_1, \ldots, j_m \in [\mathcal{N}] : (\mathsf{Init}_i \Rightarrow \varphi), \end{aligned}$$

which is an LH-assertion.

From the definition of discrete transitions $\mathcal{D}^{\mathcal{N}}$ (recall Section 2.4.3), condition (B) can be

written:

$$(\Gamma(\mathcal{N}) \wedge (\exists h \in [\mathcal{N}] : \exists \mathsf{t} \in \mathsf{Trans}_h : \mathbf{grd}(\mathsf{t}, h) \ \wedge \ \mathbf{eff}(\mathsf{t}, h))$$
$$\wedge \forall j \in [\mathcal{N}] : \ j \neq h \Rightarrow \mathbf{id}(j)) \Rightarrow \Gamma(\mathcal{N})'.$$

Here, $\mathbf{id}(i)$ is a shorthand for the formula $\bigwedge_{x[i] \in \mathsf{V}_i} x[i]' = x[i]$, and $\Gamma(\mathcal{N})'$ is the formula obtained by replacing each variable in $\Gamma(\mathcal{N})$ with its primed version. Moving the quantifier to the front and rearranging we obtain:

$$\forall h, \mathsf{t} : \exists j : (\Gamma(\mathcal{N}) \ \wedge \ \mathbf{grd}(\mathsf{t}, h) \ \wedge \ \mathbf{eff}(\mathsf{t}, h) \wedge \ (j \neq h \Rightarrow \mathbf{id}(j)) \Rightarrow \Gamma(\mathcal{N})').$$

Exposing the quantifiers in $\Gamma(\mathcal{N})$ and $\Gamma(\mathcal{N})'$:

$$\forall h, \mathsf{t} : \exists k : ((\forall i_1, \ldots, i_k : \exists j_1, \ldots, j_m : \varphi) \ \wedge \ \mathbf{grd}(\mathsf{t}, h) \ \wedge \ \mathbf{eff}(\mathsf{t}, h) \wedge$$
$$(j \neq h \Rightarrow \mathbf{id}(j)) \Rightarrow (\forall \ i'_1, \ldots, i'_k : \exists \ j'_1, \ldots, j'_m : \varphi')).$$

Moving quantifiers to the front of the formula across the implication, we obtain:

$$\forall h, \mathsf{t}, j_1, \ldots, j_m, i'_1, \ldots, i'_k : \exists j, i_1, \ldots, i_k, j'_1, \ldots, j'_m :$$
$$((\varphi \ \wedge \ \mathbf{grd}(\mathsf{t}, h) \ \wedge \ \mathbf{eff}(\mathsf{t}, h) \wedge \ (j \neq h \Rightarrow \mathbf{id}(j)))) \Rightarrow \varphi').$$

As t is universally quantified over the finite set of transitions Trans_i, it is removed by writing the above as a finite conjunction, and is an LH-assertion.

Finally, by the definition of trajectories $\mathcal{T}^{\mathcal{N}}$, condition (C) can be written as:

$$\Gamma(\mathcal{N}) \wedge (\exists t_e \in \mathbb{R} : \forall h \in [\mathcal{N}], t_p \in \mathbb{R} : \exists \mathsf{m} \in \mathsf{Loc} : \mathbf{inv}(\mathsf{m}, h) \wedge (\mathbf{stop}(\mathsf{m}, h) \Rightarrow t_2 = t_1)$$
$$\bigwedge_{x \in \mathsf{X}[h]} x[h] + t_p \mathbf{lflowrate}(\mathsf{m}, h, x) \leq x[h]' \leq x[h] + t_p \mathbf{uflowrate}(\mathsf{m}, h, x))$$
$$\Rightarrow \Gamma(\mathcal{N})', \tag{5.5}$$

where **inv** and **stop** have all continuously updated real array variables replaced with primed

versions using a time-elapse of t_p, and $\mathsf{X}[h]$ are the continuously updated real local variables. The conversion of Equation 5.5 to an LH-assertion is essentially the same as the discrete case, but more tedious. The easiest way to see this is first to convert to prenex normal form (PNF). In summary, these derivations show how we can check inductive invariants as LH-assertions for networks of hybrid automata using the small model result introduced above.

5.4 Summary

In this chapter, we develop a small model theorem for parameterized networks of hybrid automata. We use this theorem to establish inductive invariant properties for several case studies in Chapter 7. To the best of our knowledge, this is the first positive result on automatic parameterized verification of hybrid automata, beyond previous results for timed automata [32, 33, 47–50, 92, 94, 106]. The modeling framework and process of inductive invariant checking are amenable to automation, so we have implemented the ability to automatically check the inductive invariance conditions in the prototype software verification tool *Passel* using the SMT solver Z3. One weakness of the inductive invariance method is that the user is required to specify a strong enough inductive invariants to imply the desired safety property—for instance, mutual exclusion in Fischer is not an inductive invariant, and safe separation in SATS is not either—so one must find a stronger property that implies the desired safety property and is an inductive invariant). While *Passel* aids the user in this task by providing counterexample models, she/he must still manually perform the strengthening, so methods to automatically generate or strengthen invariants—such as invisible invariants [41–43, 45, 46]—or k-induction [163, 164] would be useful. We explore finding invariants automatically using an extension of the invisible invariants approach in Chapter 6.

Chapter 6

Finding Inductive Invariants

In this chapter, we present methods for finding candidate inductive invariants for parameterized networks of hybrid automata. The invariant synthesis method generates quantified inductive invariants by transforming the set of reachable states of finite instantiations of the network. This is an extension to hybrid automata of the invisible invariants method for synthesizing inductive invariants for parameterized networks of discrete automata [41, 42]. We use this extended method in a fixed-point procedure we use to generate inductive invariants of a certain class of assertions for the parameterized network of hybrid automata. These candidate inductive invariants are then checked using the method described in Chapter 5, and prove the safety property of interest if the inductive invariants imply the safety property.

6.1 Introduction

For finding candidate inductive invariants for hybrid networks, our approach builds upon the invisible invariant method used for discrete transition systems [41–43, 46, 165]. The invisible invariants method combines the standard inductive invariance proof method—recall Definition 2.4—with reachability computations to automatically perform uniform verification of safety, that is, to prove a safety property $\zeta(\mathcal{N})$ for any network $\mathcal{A}^{\mathcal{N}}$ of any size $\mathcal{N}$ (Definition 2.5). The invisible invariant method starts by computing the set of reachable states for a small instantiation of the network. Say for $\mathsf{N} = 3$, the reach set (or its approximation) for the network $\mathcal{A}^3 \triangleq \mathcal{A}(1)\|\mathcal{A}(2)\|\mathcal{A}(3)$ is computed. Then this set is projected onto a smaller instance of size $\mathsf{P} < \mathsf{N}$. Finally, this projected subset is generalized to produce a candidate invariant for a network of arbitrary size. The user's choice of P determines the shape of the generated invariant. For $\mathsf{P} = 1$ the invariant asserts properties about the variables of

a single automaton, for $\mathsf{P} = 2$ the properties may include linear inequalities involving pairs of automata, and so forth. In our methodology, the user may choose the projection to be made onto a subset of the variables of the automata in the P-sized network, such as only the real or discrete variables. This choice proves to be crucial in some of the case studies. If the generated candidate invariant is inductive and sufficient to prove some safety property $\zeta(\mathcal{N})$, then a completely automatic inductive invariance proof is obtained.

The project-generalize method is incomplete even for discrete systems [42]. The candidate invariants generated by our method may not be inductive nor are they guaranteed to prove $\zeta(\mathcal{N})$. However, we use the project-generalize method as a subroutine in a fixed-point algorithm that is guaranteed to terminate with either an automated proof of $\zeta(\mathcal{N})$ or a potential counterexample as to why $\zeta(\mathcal{N})$ may be violated. The algorithm may generate an inductive invariant with a user chosen number (P) of universally quantified automata indices for arbitrarily large networks. We implement the algorithm in *Passel* to yield the first fully automatic proof of correctness for several nontrivial hybrid networks. Notable among these are the core of SATS air-traffic control protocol [2, 26, 27], as well as Fischer's mutual exclusion protocol with drifting clocks.[1] The synthesis method finds non-trivial invariants that imply collision avoidance in SATS and mutual exclusion in Fischer.

We overcame several conceptual and technical challenges to extend the invisible invariant method to networks of hybrid automata. First, unlike the methods for discrete models [41–43, 46, 165] which primarily use BDDs for representing states—as implemented in TLV [166]—we require a symbolic representation for expressing multiple types of variables and state updates involving real arithmetic for modeling real-valued variables and their continuous evolution. In *Passel*, the states, transitions, and continuous trajectories are represented using *Passel* assertions represented as satisfiability modulo theories (SMT) formulas. This is made possible by our observation that for rectangular hybrid automata with convex invariants and stopping conditions, the (possibly nondeterministic) trajectories can be encoded by transition rules that involve a finite number of existentially quantified real-valued variables. With this representation, checking the inductive invariance conditions (Defini-

[1] Previous parameterized verification of Fischer's protocol assumed clocks evolving at unit rate $\dot{x} = 1$ [32, 47, 49], while we model rectangular dynamics $\dot{x} \in [1 - \rho, 1 + \rho]$.

tion 2.4) is done with satisfiability queries, which we perform in *Passel* using the Z3 SMT solver [136]. *Passel* uses the hybrid automata model checker PHAVer [39] for computing the reach sets for finite instances of the network, although it can also be accomplished using the SMT-based reachability method we present in Chapter 4. Second, the reach set of a finite instance is encoded in a disjunctive normal form (DNF) formula that grows exponentially with the number of instances in the network, as well as with the number of discrete locations and continuous variables of each automaton component. Naively checking satisfiability of these formulas becomes infeasible beyond the simplest of examples. To overcome this, we exploit logical equivalences—such as existential quantification distributing over disjunction—to decompose the problem into smaller, equivalent representations of the reach set encoding. This makes it possible to compute projections and generalizations of different pieces of the invariant separately, which are then combined together in a final step.

In the following sections, N and P are fixed natural numbers with $\mathsf{P} < \mathsf{N}$ and $\mathsf{N} \geq 2$ (e.g., $\mathsf{P} = 2$, $\mathsf{N} = 3$), and we recall $\mathcal{N}$ is a symbol denoting an arbitrary natural number.

6.2 Synthesizing Inductive Invariants with the Project-and-Generalize Subroutine

This section describes methods for synthesizing inductive invariants for parameterized networks of hybrid automata. If a safety property $\zeta(\mathcal{N})$ itself is not an inductive invariant for $\mathcal{A}^{\mathcal{N}}$, as is often the case, then we attempt to find stronger inductive invariants that imply $\zeta(\mathcal{N})$. We recall the general inductive invariance proof methodology from Chapter 5 of Figure 5.1, where either a user must supply a sufficiently strong candidate inductive invariant to prove a safety property, or the verification tool may try to come up with one. In this section, we first present the project-and-generalize subroutine, and then an algorithm that uses it for synthesizing inductive invariants.

The project-and-generalize subroutine is shown in Figure 6.1. This subroutine takes two input parameters N and P. The subroutine first computes the reachable states of a network $\mathcal{A}^{\mathsf{N}}$ of size N, and then through a sequence of transformations generates a candidate inductive invariant with P universally quantified index variables for a network $\mathcal{A}^{\mathcal{N}}$ of arbitrary size $\mathcal{N}$.

```
    function projectAndGeneralize(A^N, θ(N,P), N, P) {
2     V ← ∪_{i∈[N]\[P]} V_i
      R ← Reach(A^N, θ(N,P)) // assume in DNF: R = r_1 ∨ r_2 ∨ ...
4     foreach r in R {
        // project onto variables of processes 1,2,...,P
6       QF[r] ← QuantElim(∃V : r)
        // syntactically substitute 1,2,...,P to symbols i_1,...,i_P
8       foreach n in {1,2,...,P} {
          QF[r] ← Substitute(QF[r], "n", "i_n")
10      }
        // abstract index-valued variable valuations that are > P
12      foreach variable v in V_i with type(v) = [N]_⊥ {
          foreach n in {P+1, P+2, ..., N} {
14          QF[r] ← Substitute(QF[r], "v = n", "v ≠ i_1 ∧ ... ∧ v ≠ i_P")
          }
16      }
      }
18    ψ(i_1,...,i_P) ← ⋁_{r∈R} QF[r]
      return ψ(i_1,...,i_P)
20  }
```

Figure 6.1: Inductive invariant synthesis subroutine. The input arguments are the network $\mathcal{A}^{\mathsf{N}}$ (previously composed from the specification $\mathcal{A}(\mathsf{N}, i)$), a formula θ describing an initial set of states, a constant integer N, and a constant integer P, where $\mathsf{N} > \mathsf{P}$. The method computes the set of reachable from θ for a network of N automata, then transforms this reach set into an assertion $\psi(i_1, \ldots, i_{\mathsf{P}})$ over the variables of automata with (symbolic) indices i_1, i_2, ..., i_{P}.

Reachability Computation (line 3). The reach set $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$ or its overapproximation is computed for the hybrid network $\mathcal{A}^{\mathsf{N}}$ with N automata. For general hybrid automata, computing the exact reach set is undecidable, however, there are several tools available for computing bounded-time overapproximations like HyTech [37], PHAVer [39], or SpaceEx [40]. This step can use any such tool. In the results presented in this dissertation (see Section 7.8), *Passel* uses PHAVer [39], however, it also supports the SMT-based reachability approach we present in Chapter 4. The output of this step is $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$ as a disjunctive normal form (DNF) formula over the variables of $\mathcal{A}(1)$, ..., $\mathcal{A}(\mathsf{N})$.

Assumption 6.1 *For a given hybrid automaton template $\mathcal{A}(i)$ and natural number N, the reachability computation of the network $\mathcal{A}^{\mathsf{N}}$ (Definition 2.2) at line 3 terminates and is exact, yielding $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$.*

```
    function synthesis(𝒜(N, i), Init(N), N, P) {
2     𝒜 ← 𝒜_1 ‖ 𝒜_2 ‖ ... ‖ 𝒜_N
      θ(N, P) ← Init(N)
4     θ_old(N, P) ← ⊥
      // fixed-point check
6     while θ(N, P) ⇒ θ_old(N, P) is valid {
        ψ(i_1, ..., i_P) ← projectAndGeneralize(𝒜^N, θ(N, P), N, P)
8       θ_old(N, P) ← θ(N, P)
        θ(N, P) ← ∀ i_1, ..., i_P ∈ [N] : ψ(i_1, ..., i_P)
10    }
      return ψ(i_1, ..., i_P)
12  }
```

Figure 6.2: Inductive invariant synthesis fixed-point method. The input arguments are an automaton specification $\mathcal{A}(\mathcal{N}, i)$, an initial condition assertion Init, a constant natural number N, and a constant natural number P, where $\mathsf{P} < \mathsf{N}$. The fixed-point computation starts with the initial states specified by Init, then iteratively computes and transforms the reachable states of the network $\mathcal{A}^{\mathsf{N}}$ to a fixed-point. The output of the method is a candidate inductive invariant $\psi(i_1, \ldots, i_{\mathsf{P}})$.

Projection of $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$ (loop lines 4 through 17). The loop iterates over each clause $r \in \mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$.[2] Given a clause r in $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$, we project away the variables of any automata with indices greater than P. Recall that P specifies the number of universally quantified index variables in the invariant to be synthesized. *Passel* computes the projection using quantifier elimination procedures—represented by function $\mathsf{QuantElim}$ (line 6)—over the types of the variables V_i. These formulas (predicates over Booleans, linear real arithmetic, bounded integers, and their combinations) admit quantifier elimination. Based on the value of P, the quantifier elimination on line 6 is applied to $QF[r] \triangleq \exists \cup_{i \in [\mathsf{N}] \setminus [\mathsf{P}]} \mathsf{V}_i : r$, which projects away the variables of all automata with indices higher than P. In general, *Passel* projects away some subset of the variables V_i, for example, onto only the variables with discrete types or real types.

For example, in $\mathsf{Fischer}$, one of the clauses in $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$ is:

$$r \triangleq (q[1] = \mathsf{wait} \wedge q[2] = \mathsf{wait} \wedge -5 \leq -x[1] + x[2] \leq 0 \wedge g = 2 \wedge x[2] \geq 0). \tag{6.1}$$

[2]Since existential quantification distributes over disjunction, we consider each clause at a time.

For $\mathsf{P} = 1$, after executing $\mathsf{QuantElim}$, the variables of automaton 2 are eliminated to yield:[3]

$$QF[r] \triangleq \exists \mathsf{V}_2 : r \equiv \exists q[2] \in \mathsf{L}, \exists x[2] \in \mathbb{R} : r \equiv (q[1] = \mathsf{wait} \wedge 5 \geq x[1] \geq 0 \wedge g = 2). \tag{6.2}$$

Generalization of Projected Clauses (lines 8 through 16). Next, the $\mathsf{Substitute}$ function syntactically substitutes expressions in $QF[r]$. The generalization syntactically replaces all valuations of index variables equal to a value in $[\mathsf{P}]$ with fresh index symbols $i_1, i_2, \ldots, i_{\mathsf{P}}$ (lines 8 through 10). Continuing with the r from the Equation 6.2 example, the index 1 is replaced with i_1 yielding:

$$QF[r] \triangleq (q[i_1] = \mathsf{wait} \wedge 5 \geq x[i_1] \geq 0 \wedge g = 2). \tag{6.3}$$

The valuations of index-valued variables in $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$ that exceed P are transformed to be not equal to any of the symbols $i_1, \ldots, i_{\mathsf{P}}$ (lines 12 through 16).[4] In the example, $QF[r]$ has index 2 for valuations of the index-valued global variable g after projection and replacing 1 with i_1. We abstract such valuations by looking at each index-valued variable v, if the valuation $v = k$ where $k \leq \mathsf{P}$, then set $v = i_k$, and otherwise for $k > \mathsf{P}$ or $k = \perp$, set $v \neq i_1 \wedge \ldots \wedge v \neq i_{\mathsf{P}}$. Continuing the example r from Equation 6.3, we have:

$$QF[r] \triangleq (q[i_1] = \mathsf{wait} \wedge 5 \geq x[i_1] \geq 0 \wedge g \neq i_1),$$

which contains symbolic indices $i_1, \ldots, i_{\mathsf{P}}$, but no numerical indices.

Combining Clauses. Following these transformations of all r's, we take the disjunction of $QF[r]$ for all $r \in R$ (line 18). This is the formula $\psi(i_1, \ldots, i_{\mathsf{P}})$. A quantified formula is then created as (Figure 6.2, line 9):

$$\theta(\mathsf{N}, \mathsf{P}) = \dot{\forall} i_1, i_2, \ldots, i_{\mathsf{P}} \in [\mathsf{N}] : \psi(i_1, i_2, \ldots, i_{\mathsf{P}}),$$

[3] We note that $q[2]$ and $x[2]$ are constants for the quantifier elimination and not functions mapping indices to their types, as otherwise this would fall into second-order logic.

[4] We assume only equalities over any index typed variables appear in r, which is not a restriction since $[\mathsf{N}]_\perp$ is a finite set.

where $\dot{\forall}$ indicates that all the quantified indices are distinct. Since N is a finite number (e.g., 3), we could convert $\theta(\mathsf{N}, \mathsf{P})$ to a conjunction. However, for an arbitrary $\mathcal{N}$ like in the inductive invariance checks in Figure 5.1, *Passel* uses the quantifiers as in Figure 5.1, line 4.

Summary. In summary, each iteration of the loop in Figure 6.2, lines 6 through 10, which calls the function projectAndGeneralize, computes the reach set $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$—a subset of the state-space Q^{N}—then projects this onto a smaller state-space Q^{P}, and then lifts this back to Q^{N}. Although our description above is in terms of the syntactic objects and logical formulas, these operations can be described in terms of mappings between the subsets of Q^{N} and Q^{P}. We reason about monotonicity of this procedure in terms of these mappings. With N fixed for Figure 6.1, projectAndGeneralize defines a mapping $f : Pow(Q^{\mathsf{N}}) \to Pow(Q^{\mathsf{N}})$ that takes a set of states $[\![\theta(\mathsf{N}, \mathsf{P})]\!]$ and returns a set of states $[\![\psi(i_1, \ldots, i_{\mathsf{P}})]\!]$.

Proposition 6.2 *Let $f : Pow(Q^{\mathsf{N}}) \to Pow(Q^{\mathsf{N}})$ be the mapping corresponding to the operations of* projectAndGeneralize. *Under the set inclusion partial order ($\subseteq$), f is monotonic.*

Proof: We show for any $\mathbf{x}, \mathbf{y} \subseteq Q^{\mathsf{N}}$, if $\mathbf{x} \subseteq \mathbf{y}$, then $f(\mathbf{x}) \subseteq f(\mathbf{y})$. We prove this for each of the four operations that constitute projectAndGeneralize—that is, the reachability computation (line 3), the projection (line 6), the abstraction of index-valued variables (line 12), and the generalization (line 12).

Fix some $\mathbf{x} \subseteq Q^{\mathsf{N}}$ and $\mathbf{y} \subseteq Q^{\mathsf{N}}$ satisfying $\mathbf{x} \subseteq \mathbf{y}$. First, the reachability computation (line 3) defines a mapping $\pi : Pow(Q^{\mathsf{N}}) \to Pow(Q^{\mathsf{N}})$, where $\pi(\mathbf{x}) = \mathsf{Reach}(\mathcal{A}^{\mathsf{N}}, \mathbf{x})$, and we have $\pi(\mathbf{x}) \subseteq \pi(\mathbf{y})$. Most common algorithms computing (sound) overapproximations of $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$ are also monotonic.

Next, we consider some clause $r \in R$ (from the loop on line 4), where R is the DNF representation of $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}}, .)$. That is, $[\![r]\!] \subseteq \mathsf{Reach}(\mathcal{A}^{\mathsf{N}}, .) \subseteq Q^{\mathsf{N}}$. The projection (line 6) defines a mapping $\rho : Pow(Q^{\mathsf{N}}) \to Pow(Q^{\mathsf{P}})$, where $\mathsf{P} < \mathsf{N}$. That is, for any $r' \Rightarrow r$, $[\![r']\!] \subseteq [\![r]\!] \subseteq Q^{\mathsf{N}}$ and $[\![\rho(r')]\!] \subseteq [\![\rho(r)]\!] \subseteq Q^{\mathsf{P}}$, and hence ρ is monotonic. Next, the abstraction of index-valued variables in line 12 defines a mapping $\alpha : Pow(Q^{\mathsf{P}}) \to Pow(Q^{\mathsf{P}})$. Since α substitutes explicit values of index-valued variables, for instance, $v = n$ for some $n > \mathsf{P}$ with symbolic $v \neq i_1 \wedge \ldots v \neq i_{\mathsf{P}}$, it maps to an equal or larger set of states by considering every possible

valuation of i_1 through i_{P}, so α is monotonic. Since each transformation in the loop line 4 is monotonic for each clause r in R, the transformation of R is also monotonic. Finally, consider the generalization on line 9, which defines a mapping $\gamma : Pow(Q^{\mathsf{P}}) \to Pow(Q^{\mathsf{N}})$. For any $\mathbf{x}, \mathbf{y} \subseteq Q^{\mathsf{P}}$, if $\mathbf{x} \subseteq \mathbf{y}$, then $\gamma(\mathbf{x}) \subseteq \gamma(\mathbf{y})$ is satisfied, since γ is just the identity mapping (modulo renaming). Since all operations are monotone under the set inclusion partial order and the compositions of monotonic functions are monotonic, the composition, $f = \gamma \circ \alpha \circ \rho \circ \pi$ is monotonic. ■

Next, we state that projectAndGeneralize terminates.

Proposition 6.3 *Under Assumption 6.1, the function* projectAndGeneralize *terminates.*

Proof: Under Assumption 6.1, so line 3 terminates and returns a formula R describing the set of reachable states $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$ in disjunctive normal form (DNF). The reach set R is a disjunction of conjuncts, of which there are a finite number, so line 4 is called at most a finite number of times. Next, each iteration of all the operations in the loop line 4 terminate. Real arithmetic and Boolean algebras admit quantifier elimination, as do their combinations, so line 6 terminates [167–170]. Each of the loops starting with lines 8 and 12 involves a finite number of syntactic manipulations (substitutions) of finite-length formulas, and hence each terminate. Finally, line 9 involves syntactic manipulations (quantification) of finite-length formulas, and hence terminates. ■

6.3 Project-and-Generalize Example

In this section, we go through the synthesis procedure using the TMUX example from Figure 6.3. Suppose we want to synthesize inductive invariants with $\mathsf{P} = 1$ universally quantified index variables of the form $\psi(\mathcal{N}) = \forall i_1 \in [\mathcal{N}] : \phi(i_1)$, where $\phi(i_1)$ is a quantifier-free formula over the variables V_{i_1}. We accomplish this by computing reachability of finite instantiations N, and transforming this set into an inductive invariant for any $\mathcal{N} \in \mathbb{N}$. First, the reach set $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$ of a network with a finite fixed number $\mathsf{N} > \mathsf{P} \in \mathbb{N}$ of automata is computed

```
automaton name='TMux'
  // local variables
  variable name=q[i] type='L' // control location
  variable name=x[i] type='real' // continuous variable
  // global variables
  variable name=g type='index' // global lock

  location name='rem'
    flowrate: x[i]_dot >= 1.0 and x[i]_dot <= 2.0
  location name='try'
    flowrate: x[i]_dot >= 3.0 and x[i]_dot <= 4.0
  location name='cs'
    flowrate: x[i]_dot >= 5.0 and x[i]_dot <= 6.0

  transition from='rem' to='try'
    grd: g = ⊥ and x[i] >= 5.0
    ugrd: x[j] >= 10.0
    eff: x[i]' = 0.0 and g' = i
  transition from='try' to='cs'
    grd: g = i and x[i] >= 10.0
    eff: x[i]' = 0.0
  transition from='cs' to='rem'
    grd: x[i] >= 15.0
    eff: g' = ⊥ and x[i]' = 0.0

property: forall i j ((i != j and q[i] = cs) implies (q[j] != cs))
initially: forall i (q[i] = rem and x[i] = 0 and g = ⊥)
```

Figure 6.3: *Passel* input file specifying $\mathcal{A}(\mathcal{N}, i)$ for simple mutual exclusion algorithm TMUX.

(line 3).[5] For the example, we denote local variables using underscore notation instead of brackets for brevity, e.g., $x[1]$ is x_1. For the example, suppose $\mathsf{N} = 2$ so we have a formula $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$ corresponding to the reach set:

$$\begin{aligned}
\mathsf{Reach}(\mathcal{A}^{\mathsf{N}}) \triangleq\ & (q_1 = \mathsf{rem} \wedge q_2 = \mathsf{rem} \wedge g = \bot \wedge x_1 \geq 0 \wedge x_2 \geq 0) \vee \\
& (q_1 = \mathsf{try} \wedge q_2 = \mathsf{rem} \wedge g = 1 \wedge 4x_2 \geq x_1 + 40 \wedge x_1 \geq 0) \vee \qquad (6.4) \\
& (q_1 = \mathsf{rem} \wedge q_2 = \mathsf{try} \wedge g = 2 \wedge x_2 \geq 0 \wedge 4x_1 \geq x_2 + 40) \vee \qquad (6.5) \\
& (q_1 = \mathsf{rem} \wedge q_2 = \mathsf{cs} \wedge g = 2 \wedge x_2 \geq 0 \wedge 6x_1 \geq x_2 + 75) \vee \\
& (q_1 = \mathsf{cs} \wedge q_2 = \mathsf{rem} \wedge g = 1 \wedge 6x_2 \geq x_1 + 75 \wedge x_1 \geq 0).
\end{aligned}$$

Note that $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$ is in disjunctive normal form (DNF).

Next, for each conjunct r in $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$, we project away the variables of any automata with indices greater than $\mathsf{P} = 1$, since we want to synthesize an inductive invariant with

[5] For hybrid automata with dynamics specified by general (e.g., nonlinear or even linear) ordinary differential equations, this may be undecidable or yield an overapproximation of the reach set.

P (one) universally quantified index variables (line 6).[6] This projection can be computed using quantifier elimination procedures over the types of the variables V_i, and is how we accomplish this in *Passel*. For example, suppose r is from Equation 6.4, that is,

$$r \triangleq (q_1 = \mathsf{try} \wedge q_2 = \mathsf{rem} \wedge g = 1 \wedge 4x_2 \geq x_1 + 40 \wedge x_1 \geq 0).$$

Next, by projecting away the variables of automaton 2 at line 6, we have:

$$\begin{aligned} QF[r] &= \exists \mathsf{V}_2 : r \\ &= \exists q_2 \in \mathsf{L}, \exists x_2 \in \mathbb{R} : r \\ &= (q_1 = \mathsf{try} \wedge g = 1 \wedge x_1 \geq 0). \end{aligned}$$

Based on the value of P, the projection on line 6 is:

$$QF[r] \triangleq \exists \mathsf{V}^{\mathsf{N}} \setminus (\bigcup_{i \in [\mathsf{P}]} \mathsf{V}_i) : r,$$

which projects away the variables of all automata with indices higher than P when eliminating V^{N}. In general, we project away some subset of the variables V^{N}, for example, onto all the real or discrete variables. We have found in practice that it is useful to project away all but variables except the discrete ones (variables with types L and $[\mathsf{N}]_\perp$), only the control location variables and real variables, and combinations of these with and without projecting any global variables away. *Passel* uses all these heuristics.

Next, we syntactically manipulate $QF[r]$ in order to determine a quantified assertion for any choice of the number of automata $\mathcal{N}$. We define the generalization by syntactically replacing all valuations of index variables equal to a value in $[\mathsf{P}]$ with fresh index variables (symbols) $i_1, i_2, \ldots, i_{\mathsf{P}}$ (line 8). For the TMux example, we replace 1 with i_1 yielding:

$$QF[r] \triangleq (q_{i_1} = \mathsf{try} \wedge g = i_1 \wedge x_{i_1} \geq 0).$$

[6]Since existential quantification distributes over disjunction, we consider each conjunct at a time. This is one difference in our method from the original invisible invariants methods, and since that approach was implemented using BDDs, all of these operations were done on the whole reach set, whereas our representation of state is through formulas over Booleans, (bounded) integers, and reals.

For this $QF[r] \in \mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$, we are finished. However, $QF[r]$ may still contain index two for valuations of the index-valued global variable g after replacing one with i_1, for instance in Equation 6.5:

$$\begin{aligned} r &\triangleq (q_1 = \mathsf{rem} \wedge q_2 = \mathsf{try} \wedge g = 2 \wedge x_2 \geq 0 \wedge 4x_1 \geq x_2 + 40), \\ QF[r] &\triangleq (q_{i_1} = \mathsf{rem} \wedge g = 2 \wedge x_{i_1} \geq 10.0). \end{aligned}$$

Thus, we must take additional care in generalizing any index-valued variables (line 12). Since some of the valuations of index-valued variables in $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$ will exceed P (since $\mathsf{P} < \mathsf{N}$), we must transform these valuations to symbols equal (or not equal) to $i_1, \ldots, i_{\mathsf{P}}$. The process described by line 12 is: for any index-valued variable v, if the valuation $v = k$ where $k \leq \mathsf{P}$, then set $v = i_k$, and otherwise for $k > \mathsf{P}$ or $k = \perp$, set $v \neq i_1 \wedge \ldots \wedge v \neq i_{\mathsf{P}}$. For the TMux example for r from Equation 6.5, we have:

$$QF[r] \triangleq (q_{i_1} = \mathsf{rem} \wedge g \neq i_1 \wedge x_{i_1} \geq 10.0),$$

which is now ensured not to contain any indices or index-valued variable valuations other than the symbols i_1, ..., i_{P}.

Finally, we take the disjunction of the $QF[r]$'s for all $r \in R$ (line 18). For the TMux example, this yields the assertion:

$$\begin{aligned} \phi(i_1) \triangleq\ & (q_{i_1} = \mathsf{rem} \wedge g \neq i_1 \wedge x_{i_1} \geq 0) \vee \\ & (q_{i_1} = \mathsf{rem} \wedge g \neq i_1 \wedge x_{i_1} \geq 10.0) \vee \\ & (q_{i_1} = \mathsf{rem} \wedge g \neq i_1 \wedge x_{i_1} \geq 12.5) \vee \\ & (q_{i_1} = \mathsf{try} \wedge g = i_1 \wedge x_{i_1} \geq 0) \vee \\ & (q_{i_1} = \mathsf{cs} \wedge g = i_1 \wedge x_{i_1} \geq 0). \end{aligned}$$

Finally, a quantified formula $\theta(\mathsf{N}, \mathsf{P})$ (Figure 6.2, line 9) is created as:

$$\theta(\mathsf{N}, \mathsf{P}) = \dot{\forall} i_1, i_2, \ldots, i_{\mathsf{P}} \in [\mathsf{N}] : \psi(i_1, i_2, \ldots, i_{\mathsf{P}}),$$

where $\dot{\forall}$ indicates that all the quantified index variables are distinct (i.e., $i_1 \neq i_2 \ldots \neq i_{\mathsf{P}}$). At this point, since N is a fixed, finite number (e.g., 3), we could convert $\theta(\mathsf{N}, \mathsf{P})$ to a conjunction, but for an arbitrary $\mathcal{N}$, we would need the quantifiers. To summarize, projectAndGeneralize computes the reach set—a subset of the state-space Q^{N}—then projects this onto a smaller state-space Q^{P}, and then lifts back to Q^{N}, with the hope that it will be an inductive invariant in $Q^{\mathcal{N}}$ for networks $\mathcal{A}^{\mathcal{N}}$ of arbitrary size $\mathcal{N}$.

6.4 Inductive Invariant Synthesis Fixed-Point Procedure

In this section, we present an algorithm (see Figure 6.2) that uses projectAndGeneralize for synthesizing inductive invariants of hybrid automata networks. First, we compute the composition $\mathcal{A}^{\mathsf{N}}$ of a fixed number N of automaton $\mathcal{A}(\mathsf{N}, i)$, by first instantiating each $\mathcal{A}_1$, ... $\mathcal{A}_{\mathsf{N}}$ and then taking their composition. This composed automaton is an input argument for projectAndGeneralize (line 2). The synthesis procedure repeatedly calls projectAndGeneralize to generate assertions $\psi(i_1, \ldots, i_{\mathsf{P}})$. We fix N, P, and $\mathcal{A}^{\mathsf{N}}$, and denote the mathematical function computed by projectAndGeneralize$(\mathcal{A}^{\mathsf{N}}, \theta(\mathsf{N}, \mathsf{P}), \mathsf{N}, \mathsf{P})$ by $f(\theta(\mathsf{N}, \mathsf{P}))$ and suppress the other arguments. The first iteration of f uses the set of initial states Θ^{N} as the initial state argument of projectAndGeneralize, then subsequent iterations use the set of states corresponding to the generated assertion $\theta(\mathsf{N}, \mathsf{P})$ from line 9 as a new set of initial states. Since N is a fixed integer, the formula $\theta(\mathsf{N}, \mathsf{P}) \triangleq \dot{\forall} i_1, \ldots, i_{\mathsf{P}} \in [\mathsf{N}] : \psi(i_1, \ldots, i_{\mathsf{P}})$ from line 9 is equivalent to a finite conjunction. Once synthesis reaches a fixed-point (shown below), the final assertion is used as a candidate inductive invariant for the network composed of an arbitrary number $\mathcal{N}$ of automata $\mathcal{A}(\mathcal{N}, i)$.

We first recall Kleene's fixed-point theorem [171].

Theorem 6.4 *Let $\Lambda = (S, \subseteq)$ be a complete partial order, and let $f : \Lambda \to \Lambda$ be a monotone function. Then f has a least fixed-point, which is the least fixed-point (supremum) of the ascending Kleene chain of f, $\bot \preceq f(\bot) \preceq f(f(\bot)) \preceq \ldots \preceq f^*_{\mathsf{N},\mathsf{P}}(\bot)$.*

Here, $\Lambda = (Pow(Q^{\mathsf{N}}), \subseteq)$ is described by the subset partial order relation ($\preceq$=$\subseteq$) on the set of subsets of the state-space ($S = Pow(Q^{\mathsf{N}})$). The function $f : Pow(Q^{\mathsf{N}}) \to Pow(Q^{\mathsf{N}})$ is

described by the procedures of projectAndGeneralize, as used earlier in Proposition 6.2, where it is called repeatedly in the loop at line 6.

Proposition 6.5 *Under Assumption 6.1, the* synthesis *function (Figure 6.2) terminates and its output ψ is the least fixed-point $f^*_{\mathsf{N},\mathsf{P}}$ of the function f.*

Proof: With Assumption 6.1, by Proposition 6.3, projectAndGeneralize terminates. The partial order $\Lambda = \langle Pow(Q^{\mathsf{N}}), \subseteq \rangle$ is complete and f is monotonic under $\subseteq$ (by *Proposition* 6.2), therefore, by Kleene's fixed-point theorem (Theorem 6.4), the loop terminates and the computed ψ is the least fixed-point of f. ■

Proposition 6.6 *For a hybrid automaton network $\mathcal{A}^{\mathsf{N}}$ with initial assertion $\mathsf{Init} \triangleq \forall i \in [\mathsf{N}] : \mathsf{Init}_i$, there exists an assertion $\theta(\mathsf{N},\mathsf{P}) \triangleq \dot{\forall} i_1, \ldots, i_{\mathsf{P}} \in [\mathsf{N}] : \psi(i_1, \ldots, i_{\mathsf{P}})$ such that the least fixed-point $f^*_{\mathsf{N},\mathsf{P}}$ is $\theta(\mathsf{N},\mathsf{P})$.*

Proof: By the synthesis routine, each iteration has f with the shape $\dot{\forall} i_1, \ldots, i_{\mathsf{P}} \in [\mathsf{N}] : \psi(i_1, \ldots, i_{\mathsf{P}})$. By Proposition 6.5, $f^*_{\mathsf{N},\mathsf{P}}$ is the least fixed-point of f, and is thus the strongest assertion with the given shape that can be computed as the fixed-point of f. ■

The next proposition allows us to conclude whether $\mathcal{A}^{\mathsf{N}}$ has *any* inductive invariants of the shape generated using P quantified indices that is sufficient for proving $\zeta(\mathsf{N})$, by checking if the least fixed-point $f^*_{\mathsf{N},\mathsf{P}}$ is an inductive invariant and implies $\zeta(\mathsf{N})$.

Proposition 6.7 *Given N, $\mathsf{P} \in \mathbb{N}$, the following statements are equivalent:*

(a) there exists an inductive invariant $\theta(\mathsf{N},\mathsf{P})$ with the shape $\forall i_1, \ldots, i_{\mathsf{P}} \in [\mathsf{N}] : \psi(i_1, \ldots, i_{\mathsf{P}})$ such that $\theta(\mathsf{N},\mathsf{P}) \Rightarrow \zeta(\mathsf{N})$, and

(b) the least fixed-point $f^*_{\mathsf{N},\mathsf{P}} \Rightarrow \zeta(\mathsf{N})$.

Proof: (b) ⇒ (a): By (b), $f^*_{\mathsf{N},\mathsf{P}}(\mathsf{Init}) \Rightarrow \zeta(\mathsf{N})$, then it follows that $\theta(\mathsf{N},\mathsf{P}) \Rightarrow \zeta(\mathsf{N})$. (a) ⇒ (b): By (a), there is some $\theta(\mathsf{N},\mathsf{P})$ such that $\theta(\mathsf{N},\mathsf{P}) \Rightarrow \zeta(\mathsf{N})$. Since $f^*_{\mathsf{N},\mathsf{P}}$ is the least fixed-point, we have $f^*_{\mathsf{N},\mathsf{P}} \Rightarrow \theta(\mathsf{N},\mathsf{P})$, so we also have $f^*_{\mathsf{N},\mathsf{P}} \Rightarrow \zeta(\mathsf{N})$. ■

Up to this point, everything has been for a fixed sized network $\mathcal{A}^{\mathsf{N}}$ of N automata. If the class of systems under consideration satisfies a small model theorem (Theorem 5.2), then it is possible to state a completeness result, namely that synthesis generates an inductive invariant of a particular shape (quantifier order) for networks $\mathcal{A}^{\mathcal{N}}$ composed of an arbitrary number $\mathcal{N}$ of automata. For example, one such result appears in [43] as Theorem 5. This result can be applied to parameterized networks of initialized rectangular hybrid automata (IRHA) by converting the initialized rectangular hybrid automaton template to a finite-state automaton using the method of [30].

6.5 Summary

This chapter presents two methods for finding candidate inductive invariants for parameterized networks of hybrid automata. The first method, project-and-generalize—inspired directly from the original works on finding invariants for discrete networks using the invisible invariants method [41, 42]—computes the reachable states for small instantiations of the network, then transforms this set by projecting and generalizing it to a candidate invariant with a certain shape of quantification for the parameterized network. The method is necessarily incomplete, in that it fails to generate candidate invariants for networks even of a particular, restricted shape of quantification (for example, all universally quantified indices, $\forall i_1, i_2, \ldots$). For this reason, we use the method as a subroutine in a fixed-point synthesis procedure used to generate candidate invariants of a certain shape namely $\forall i_1, i_2, \ldots : \phi(i_1, i_2, \ldots)$. If this invariant does not prove a desired safety property, the user can manually refine the proof of inductive invariance by searching for candidates with other shapes (for example, $\exists i \forall j \exists k$). While it is possible to dualize the search (see [43]) to find candidates of the form $\exists^*$ instead of $\forall^*$, finding candidates with alternating shapes would require different techniques, although some completing techniques for discrete systems might be applicable, such as [172]. We implement the methods in the *Passel* verification tool that we describe in detail in Chapter 7. We experimentally evaluate the methods in Section 7.8, where they have been applied successfully to fully automatically prove safety for several case studies like Fischer's mutual exclusion and SATS.

Chapter 7

Passel and Experimental Evaluation

Passel—which is a collective noun meaning a large group of people or things of indeterminate number—is a software tool developed as a part of this dissertation for modeling and verifying safety properties for parameterized networks of hybrid automata. In this chapter, we describe some of the engineering and design choices made in developing *Passel*, as well as several other case studies modeled and verified using *Passel*.[1] We present promising experimental results where the invariant synthesis procedures of Chapter 6, combined with the inductive invariance checks of Chapter 5, have been useful in automatically proving safety properties of examples like Fischer's mutual exclusion protocol with rectangular dynamics instead of clocks, the SATS conceptual air-traffic control protocol, and others. We also present promising experimental results for the symmetry-reduced reachability computation of finite instantiations of networks of hybrid automata of Chapter 4.[2]

7.1 Introduction

In this chapter, we present *Passel*, a software tool that embodies the uniform verification techniques for parameterized networks of hybrid automata discussed in the previous chapters. Given a safety property $\zeta(\mathcal{N})$ parameterized by the size $\mathcal{N}$ of the network, and a rectangular hybrid automaton template $\mathcal{A}(\mathcal{N}, i)$, *Passel* attempts to verify that for any natural number $\mathcal{N}$, the parameterized network of size $\mathcal{N}$ obtained by the parallel composition $\mathcal{A}(1)\| \ldots \|\mathcal{A}(\mathcal{N})$ satisfies $\zeta(\mathcal{N})$. Rectangular hybrid automata admit continuous dynamics of the form $\dot{x} \in [a, b]$, which can exactly describe continuous variables with con-

[1] *Passel*, the specification files for the examples evaluated, and output logs illustrating anonymized representations of reachable states, inductive invariance proofs, and synthesized inductive invariants are available for download from https://publish.illinois.edu/passel-tool/.

[2] This chapter is based in part on prior work [3], portions of which are reprinted here with permission.

stant slope, such as drifting clocks. Rectangular dynamics can also approximate more complex linear and nonlinear differential equations arbitrarily closely up to some bounded time and over bounded regions of the state-space. We assume that the property to be verified $\zeta(\mathcal{N})$ is an index sentence, that is, all of the index variables are quantified (recall Section 2.3.2). For instance, collision avoidance is specified by the index sentence $\zeta(\mathcal{N}) = \forall i, j \in [\mathcal{N}] : i \neq j \Rightarrow \|x[i] - x[j]\| \geq S$, where $x[i]$,$x[j] \in \mathbb{R}^3$ and $S > 0$, and mutual exclusion is specified by $\zeta(\mathcal{N}) = \forall i, j \in [\mathcal{N}] : (i \neq j \wedge q[i] = \mathsf{cs}) \Rightarrow q[j] \neq \mathsf{cs}$.

The core of *Passel* has procedures for checking and finding inductive invariants for hybrid networks of arbitrary size, as described respectively in Chapters 5 and 6. For checking quantified inductive invariants, *Passel* uses quantifier elimination and instantiation, or exploits small model properties of the inductive invariant assertions of hybrid networks, such as Theorem 5.2. Additionally, *Passel* implements the reachability computation techniques introduced in Chapter 4 for finite instantiations of hybrid automata networks.

7.2 Overview

The inputs to *Passel* are:

(a) a syntactic description of a template hybrid automaton, $\mathcal{A}(\mathcal{N}, i)$, including its variables, discrete transitions, locations with invariants and rates, initial conditions (such as presented in Figures 2.1, 2.3, and 4.11),[3]

(b) a list of safety properties $\mathcal{P}$ that should be proved,

(c) an optional list of any parameters used in the protocols,

(d) an optional list of assumptions on the variables and parameters.

These inputs are specified in an extension of the HyXML specification language developed for specifying hybrid automata [173], as defined in Section 2.3.

Passel supports several modes of operation, including:

[3]While UPAAAL [38] allows for specifying timed automaton templates like $\mathcal{A}(\mathcal{N}, i)$ and then computes $\mathcal{A}^{\mathsf{N}}$ for a fixed N, all the hybrid systems model checkers (e.g., [37, 39, 40]) require the user to specify each $\mathcal{A}_1$, ..., $\mathcal{A}_{\mathsf{N}}$, and the model checker may then compose these.

(a) bounded model checking with respect to each property in $\mathcal{P}$ using anonymized reachability (Chapter 4),

(b) checking if each property in $\mathcal{P}$ is an inductive invariant (Chapter 5),

(c) creating a PHAVer, [39] input model for a finite instantiation of size N (for use on its own or in the invariant synthesis method of Chapter 6),[4]

(d) performing the invisible invariants [41–43,45,46] procedure using PHAVer (that is, only one iteration of the projectAndGeneralize method of Figure 6.1 from Chapter 6), and

(e) performing the invariant synthesis fixed-point computation using PHAVer (that is, using the synthesis method of Figure 6.2, which iterates projectAndGeneralize of Figure 6.1 to a fixed-point).

7.3 Implementation

The current implementation of *Passel* uses the SMT solver Z3 [136] for proving validity, checking satisfiability, and performing quantifier elimination. *Passel* is written in C# and uses the managed .NET API to version 4.2 of Z3. *Passel* proves validity of a formula ϕ by checking unsatisfiability of $\neg\phi$. First, the variables V_i used in defining $\mathcal{A}(\mathcal{N}, i)$ are specified to the SMT solver. *Passel* automatically generates and asserts trivial data-type lemmas that the SMT solver requires. Next, the list of assumptions given to *Passel* are asserted as axioms to the SMT solver.

Passel configures Z3 to use a variety of options, in particular, it requires either having model-based quantifier instantiation (MBQI) enabled or quantifier elimination enabled, as otherwise we may receive unknown as a response from Z3 for some satisfiability checks. Within the SMT solver, we model array variables of automata as uninterpreted functions mapping a subset of the integers (i.e., the set of process indices) to the type of the variable. We tried using the theory of arrays for this instead of uninterpreted functions, but the

[4]We took great care in the development and integration of *Passel* and PHAVer to ensure consistency of semantics.

performance was significantly worse. Global variables are modeled as constants of their types.

Reachability Using Anonymized States. The reachability method using anonymized states of Chapter 4 is implemented in *Passel* as a fixed-point procedure, shown in the pseudocode in Figure 4.3. The initial condition Init_i is converted to an anonymized state and added to the frontier of states, implemented as a hashset. For each anonymized state in the frontier, any enabled transition is taken, and any resulting anonymized states are added to the frontier, so long as it is not already in the set of anonymized reachable states, which is also implemented as a hashset.

Proving Inductive Invariants. *Passel* implements an automatic method for checking the conditions for inductive invariance (Definition 2.4) for parameterized networks of hybrid automata, as shown in the function inductiveInvariance in Figure 5.1. The inductive invariance conditions (Definition 2.4) are encoded using formulas appearing essentially as they do in the definitions of the discrete and continuous transition relations in Section 2.4. We did not need to use any special encoding to represent our systems for Z3, so the queries we ask are almost exactly the same as the formulas appearing in Section 5.3. Given the finite bound N_0 from Theorem 5.2, we could potentially have composed the system for each instantiation $2 \leq \mathsf{N} \leq \mathsf{N}_0$ and used existing tools (for instance, HyTech [37] or PHAVer [39]), but the LH-assertions specifications were more natural to state in an environment that allows quantifiers. Additionally, our prior experience in model checking such parameterized systems [2] indicated that the bound allowed in practice due to memory requirements may be less than the bound N_0, and may prevent verification.

Each property $\zeta(\mathcal{N}) \in \mathcal{P}$ is checked as an inductive invariant by proving—checking the unsatisfiability of—the inductive invariance conditions as formalized for parameterized networks of hybrid automata in Section 5.1.1. If each of these conditions is proved, then $\zeta(\mathcal{N})$ is an inductive invariant, and it is asserted as an axiom. Then, the process repeats with the next property in $\mathcal{P}$. The order of operation in which these properties are attempted to be proved matters. Thus, if any property is proved as an inductive invariant, then all other

properties either not processed or previously disproved will be repeated.

The operation of checking an inductive invariant is as follows. The user specifies a set $\mathcal{P}$ of candidate inductive invariants. Based on the protocol specification, we receive from Theorem 5.2 a bound N_0 on the number of automata N for which we must check each property. Many of the invariant properties we are interested in are not inductive (e.g., mutual exclusion in Fischer's protocol is not inductive, nor even k-inductive [103]), so having a set of candidate invariants allows us to discharge each until we have a set of proven lemmas that imply the desired invariant. For each candidate $\Gamma(\mathcal{N}) \in \mathcal{P}$, we check if $\Gamma(\mathcal{N})$ is an inductive invariant by attempting to prove $\Gamma(\mathcal{N})'$ after each transition, where $\Gamma(\mathcal{N})'$ is $\Gamma(\mathcal{N})$ with all variables replaced with their primed counterparts (i.e., post-states). If $\Gamma(\mathcal{N})$ is successfully proved, we assert $\Gamma(\mathcal{N})$ as a lemma and check some other candidate in $\mathcal{P}$ until we have proved—or failed to prove—each property in $\mathcal{P}$. We emphasize that if we do not prove a property, it does not necessarily mean that the property does not hold, only that it may not be inductive. Thus, if we terminate with a proof that a desired safety property $\zeta(\mathcal{N})$ holds, then it indeed holds, but if not, then we cannot conclude it is violated, only that the candidate invariants $\mathcal{P}$ were not sufficient to prove $\zeta(\mathcal{N})$. For sanity checking (of soundness), we should not be able to prove any property is an inductive invariant that is not.

The main difficulty in safety verification using inductive invariance is specifying a rich enough set of properties $\mathcal{P}$. We perform satisfiability checks with model construction enable, thus if a transition or time elapse violates the candidate property, we record it and display it to the user so she/he may use this information to refine the candidate manually. For example, for Fischer, a detailed inductive invariant refinement is performed in [103]. The set of properties $\mathcal{P}$ for Fischer is shown in Equation 2.6. However, we also had to add more properties, partly because we are working with a different semantics compared to the timeout automata considered in [103].

Finding Inductive Invariants. The inductive invariant synthesis methods that *Passel* implements (see Chapter 6 and refer to Figures 6.1 and 6.2) rely on projection and syntactic manipulations. The projection is implemented using quantifier elimination and the syntactic operations include expression replacement and quantifier introduction. We note

that we attempted to implement the invariant synthesis fixed-point method in *Passel* using the fixed-point engine built into Z3 [174, 175], but found it unsuitable for timed examples. In practice, it is useful to project away all variables—Figure 6.1, line 6—except the discrete ones (variables with types L and $[\mathsf{N}]_\perp$), only the control location variables and real variables, and combinations of these with and without projecting any global variables away, so *Passel* does this. *Passel* models the local variables of $\mathcal{A}(i)$ as unary functions, mapping indices to the variables type—for each local variable $x \in \mathsf{V}_i$, $x : [\mathcal{N}] \to \mathit{type}(x)$, where $\mathcal{N}$ is not fixed a priori, but has some assumption specified, such as $\mathcal{N} \geq 2$ or $\mathcal{N} \geq 2 \wedge \mathcal{N} \leq 73$. However, when eliminating quantifiers for the projection part, *Passel* first converts these to constants, as otherwise it would result in a second-order logic formula that the SMT solver cannot process.

7.4 Additional Examples

We have analyzed several examples in *Passel*, several of which have been presented as examples in this dissertation, such as Figures 2.1, 2.3, and 4.11. Our examples include distributed algorithms, cache coherence protocols, and other purely discrete parameterized systems. Our main purpose in developing *Passel* was for the verification of timed and hybrid parameterized systems, so we also have several of these types of examples, including timed distributed mutual exclusion algorithms like Fischer from Chapter 2 and the SATS example from Chapter 2.

The buggy version of SSATS replaces the precondition $l[\mathit{next}[i]]$ with $l[\mathit{last}]$ and $x[\mathit{next}[i]]$ with $x[\mathit{last}]$ in Figure 2.3, line 27, which ensures the spacing between i and the last aircraft is large enough. However, this may not be the aircraft immediately ahead of i, for instance, if two aircraft have moved to the base, so the safe separation properties do not hold. The properties specifying a safe separation are (SD) and (SE).

We checked property (SD) only for rectangular dynamics (that is, Figure 2.3, line 19 is as written) and (SE) only for timed dynamics (that is, the rectangular dynamics in Figure 2.3, line 19 are replaced by $\dot{x}[i] = 1$ or $\dot{x}[i] = v_{min} = v_{max}$). This is because SATS with rectangular dynamics does not satisfy (SE). The properties are:

(SA) $\forall i \in [\mathcal{N}] : q[i] = \mathsf{fly} \Rightarrow last \neq i$,

(SB) $\forall i, j \in [\mathcal{N}] : next[j] = i \Rightarrow q[i] \neq \mathsf{fly}$,

(SC) $\forall i, j \in [\mathcal{N}] : (q[i] = \mathsf{hold} \wedge next[j] = i) \Rightarrow q[j] = \mathsf{hold}$,

(SD) $\forall i, j \in [\mathcal{N}] : (q[i] = \mathsf{base} \wedge q[j] = \mathsf{base} \wedge next[j] = i) \Rightarrow x[i] \geq L_S + (v_{max} - v_{min}) \frac{L_B - x[j]}{v_{min}}$, and

(SE) $\forall i, j \in [\mathcal{N}] : (i \neq j \wedge q[i] = \mathsf{base} \wedge q[j] = \mathsf{base} \wedge next[j] = i) \Rightarrow x[i] - x[j] \geq L_S$.

7.4.1 Fischer's Mutual Exclusion Protocol with Auxiliary Variables

Figure 7.1 is a description of Fischer-Aux-Timed, which is Fischer's mutual exclusion protocol with auxiliary real-valued variables that are used to track the earliest and latest times at which a state transition can occur [107]. This style of modeling has been extensively used; see, for example, [24, 103]. The correct version of Fischer's mutual exclusion protocol has a constraint $A < B$, and the buggy version has $A \geq B$.

We checked the following properties for Fischer-Aux-Timed:

(FA) $\forall i, j \in [\mathcal{N}] : x[i] = x[j]$,

(FB) $\forall i \in [\mathcal{N}] : q[i] = \mathsf{set} \Rightarrow last[i] \leq x[i] + A$,

(FC) $\forall i \in [\mathcal{N}] : q[i] = \mathsf{set} \Rightarrow x[i] \leq last[i]$,

(FD) $\forall i, j \in [\mathcal{N}] : (q[i] = \mathsf{check} \wedge g = i \wedge q[j] = \mathsf{set}) \Rightarrow first[i] > last[j]$,

(FE) $\forall i, j \in [\mathcal{N}] : q[i] = \mathsf{cs} \Rightarrow (g = i \wedge q[j] \neq \mathsf{set})$, and

(FF) $\forall i, j \in [\mathcal{N}] : (i \neq j) \Rightarrow (q[i] \neq \mathsf{cs} \vee q[j] \neq \mathsf{cs})$,

where (FF) specifies mutual exclusion. These properties represent a manual strengthening proof of inductive invariance, where (FE) is stronger than mutual exclusion and also inductive.

```
parameter name='A' type='real' value = 5.0  // smaller timing parameter
parameter name='B' type='real' value = 35.0 // larger timing parameter
parameter name='lb' type='real' value = 1.0 // lower clock rate
parameter name='ub' type='real' value = 2.0 // upper clock rate

automaton name='Fischer-Aux'
  variable name='q[i]' type='L' // control location local variable
  variable name='x[i]' type='real' // continuous local variable
  variable name='first[i]' type='real' // first time local variable
  variable name='last[i]' type='real' // last time local variable
  variable name='g' type='index' // global lock variable

  location name='rem'
    flowrate: x[i]_dot = 0.0
  location name='try'
    inv: x[i] <= last[i]
    stop: x[i] = last[i]
    flowrate: x[i]_dot >= lb and x[i]_dot <= ub
  location name='wait'
    flowrate: x[i]_dot >= lb and x[i]_dot <= ub
  location name='cs'
    flowrate: x[i]_dot = 0.0

  transition from='rem' to='try'
    grd: g = ⊥
    eff: last[i]′ = x[i] + A
  transition from='try' to='wait'
    eff: g′ = i and first[i]′ = x[i] + B
  transition from='wait' to='cs'
    grd: g = i and x[i] >= first[i]
  transition from='wait' to='rem'
    grd: g != i and x[i] >= first[i]
    eff: first[i]′ = 0.0
  transition from='cs' to='rem'
    eff: g′ = ⊥

property: forall i j ((i != j and q[i] = cs) implies (q[j] != cs))
initially: forall i (q[i] = rem and x[i] = 0 and last[i] = A
              and first[i] = 0 and g = ⊥)
```

Figure 7.1: *Passel* input file specifying $\mathcal{A}(\mathcal{N}, i)$ for Fischer's mutual exclusion algorithm with auxiliary variables Fischer-Aux-Timed.

7.5 Experimental Setup

All experiments were conducted on a modern laptop with a 2.2 GHz quad-core Intel Core i7-2670QM processor and 16 GB RAM, running 64-bit Windows 8. *Passel* is written in *C#* and used the Z3 SMT solver version 4.1 [136] through the *C#* API. The experiments were performed under a 32-bit Ubuntu virtual machine using VMWare Player on the same laptop, with access to two cores and 4 GB RAM. *Passel* was executed using `Mono`. *Passel* takes a syntactic specification for a single template hybrid automaton $\mathcal{A}(\mathcal{N}, i)$ in a variant of HyXML [173], as described in detail in Section 2.3.

In the synthesis experiments, *Passel* used PHAVer (version 0.38) for the reachability com-

putation $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$ of a finite network $\mathcal{A}^{\mathsf{N}}$ of N automata. PHAVer was run under an Ubuntu virtual machine using VMWare Player on the same laptop, with access to two cores and 4 GB RAM. Input files for PHAVer were generated by *Passel* from the data structure inside *Passel* encoding the syntactic structure of $\mathcal{A}(i)$. We measured time and memory usage of *Passel* and PHAVer with `memtime`. For some measurements of *Passel* subroutines—like those presented for synthesis and checking inductiveness in Table 7.2—we used internal timers in *Passel* for benchmarking.

7.6 Experimental Results for Reachability Using Anonymized States

In this section, we describe the experimental results using *Passel*'s implementation of the reachability method using anonymized states of Chapter 4.

Figure 7.2 shows a runtime comparison between PHAVer and *Passel* for several examples as a function of N, the number of automata in the finite instantiation of the network. Figure 7.3 shows a memory usage comparison for the examples also as a function of N.

Nondeterministic Finite-State Automaton. The first example specifies a simple nondeterministic finite-state automaton example with 5 states and 10 transitions. This artificial example is created purely to demonstrate the strength the anonymized state representation. For the NFA example, PHAVer is only able to compute the reachable states up to $\mathsf{N} = 6$ before running out of memory due to its representation of all the permutations of reachable states. Even at $\mathsf{N} = 6$, PHAVer uses about 600 MB memory and required over 3 minutes to compute the reachable states. While $\mathcal{A}^6$ only has $5^6 = 15625$ states, PHAVer utilizes an inefficient explicit-state representation. In comparison, the reachability method from Chapter 4 implemented in *Passel* computed the reachable states for the same example in an order of magnitude less time (about 20 seconds) and used about an order of magnitude less memory (about 75 MB). Furthermore, *Passel* was able to compute the set of reachable states up to $\mathsf{N} = 30$ in a little over an hour, while using about 220 MB memory.

The memory usage for the NFA example shown in Figure 7.3 illustrates the strength of

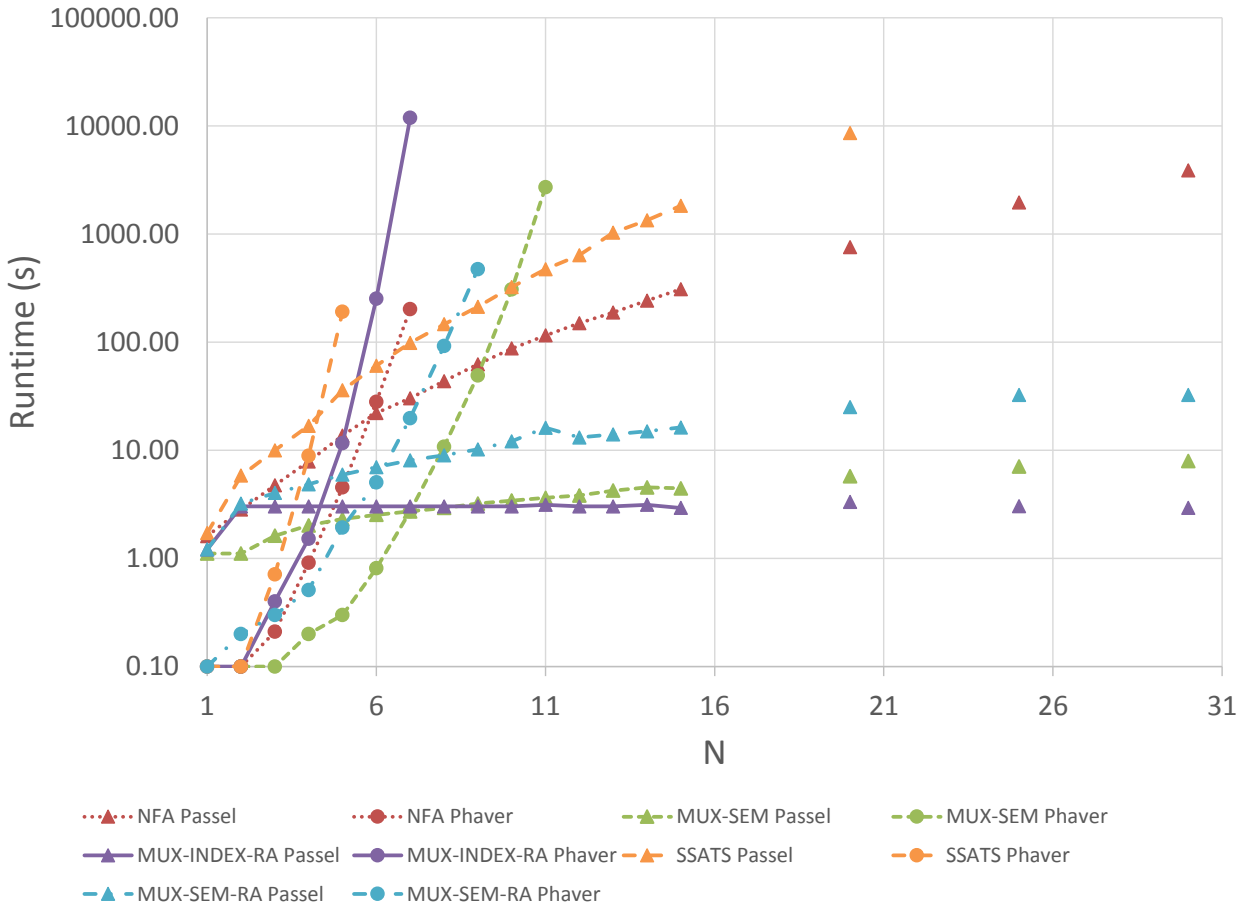

Figure 7.2: Anonymized reachability runtime comparison of PHAVer and *Passel* for several examples. The vertical axis is logarithmic and has units of seconds. The horizontal axis is the number of automata N.

Passel's anonymized reachability method. While *Passel* initially uses much more memory than PHAVer—in part due to loading a variety of libraries, including the API to Z3—its scaling as a function of N is far superior. This is highlighted by *Passel* using about a third of the memory at $N = 30$ of 220 MB compared to PHAVer's usage of over 600 MB memory at $N = 6$, even though the problem size in terms of N is 5 times larger.

MUX-SEM **Mutual Exclusion.** This is a standard mutual exclusion algorithm implemented using a semaphore, which we also use in Chapter 4 for explaining the anonymized state representation and reachability algorithm. See Figures 4.1 and 4.2 for the complete specification. To illustrate the difference between the state-space representations, at $N = 7$, *Passel* has 15 reachable discrete states represented as an anonymized state representation

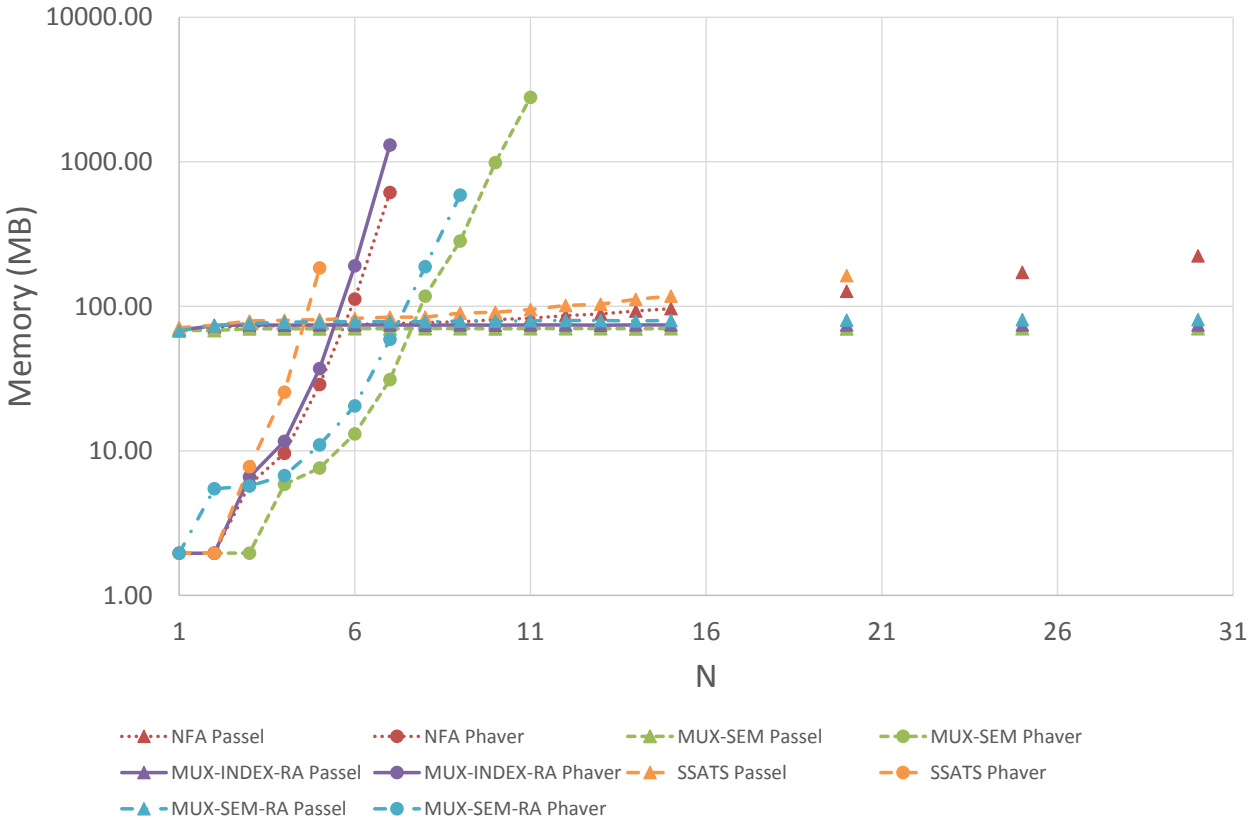

Figure 7.3: Anonymized reachability memory usage comparison of PHAVer and *Passel* for several examples. The vertical axis scale is logarithmic and has units of megabytes. The horizontal axis is the number of automata N.

S, whereas PHAVer represents this with 576 states. The entire state-space of MUX-SEM for N = 7 has 4374 discrete states, without using the anonymized representation.

For N = 11, PHAVer runs out of memory, so comparisons beyond this value are not possible. As shown in Figure 7.3, for N = 10, PHAVer uses over 2.5 GB memory and completes in about 45 minutes, while *Passel* uses more than order of magnitude less memory at about 70 MB and nearly three orders of magnitude less runtime at about 3.5 seconds.[5] Because of the anonymized representation of the state-space, *Passel* is able to compute the reachable states of N = 30 in under ten seconds (Figure 7.2) using about 70 MB memory (Figure 7.3). As shown in Figures 7.4 and 7.5, *Passel* is able to easily scale to hundreds of automata for MUX-SEM with modest runtime and memory usage.

[5]Note that Z3 has some nondeterministic heuristics built-in that cause some of the memory fluctuations seen in the *Passel* results.

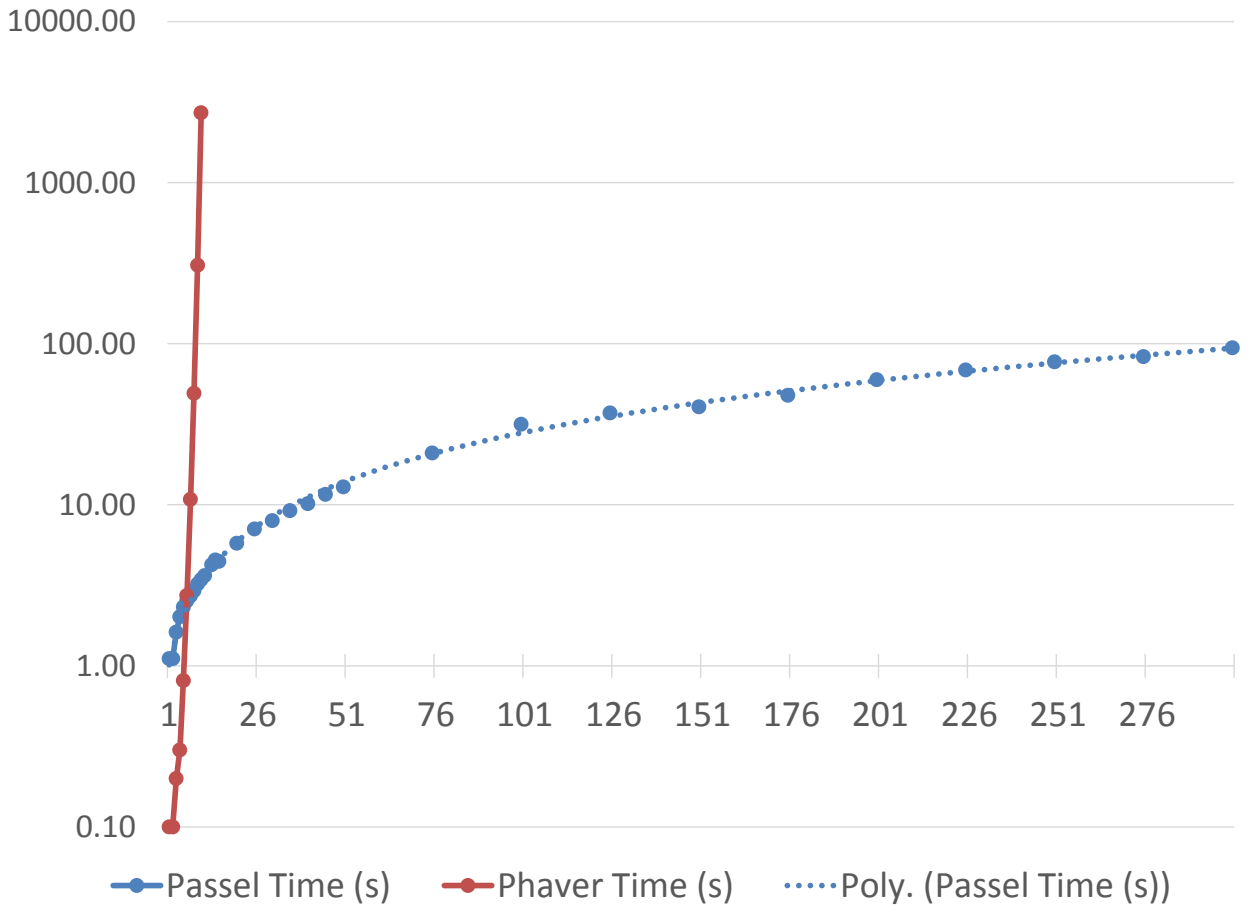

Figure 7.4: Anonymized reachability runtime comparison of PHAVer and *Passel* for MUX-SEM. The vertical axis scale is logarithmic. The horizontal axis is the number of automata N. This illustrates scaling to hundreds of automata.

MUX-INDEX-RECT **Mutual Exclusion.** For the timed mutual exclusion algorithm that we previously described in Chapter 4 (Figure 4.11). PHAVer runs out of memory for $N = 8$. As shown in Figures 7.2 and 7.3, for $N = 7$, PHAVer uses over 1.3 GB memory and completes in over 3 hours, while *Passel* uses over an order of magnitude less memory at about 70 MB and nearly four orders of magnitude less runtime at about three seconds. Because of the anonymized representation of the state-space, *Passel* is able to compute the reachable states of $N = 30$ in a few seconds using about 70 MB memory, and *Passel* is able to easily scale to thousands of automata for MUX-INDEX-RECT. This is because the number of elements in the anonymized state-space representation does not grow as a function of N, as discussed in Section 4.4.1.

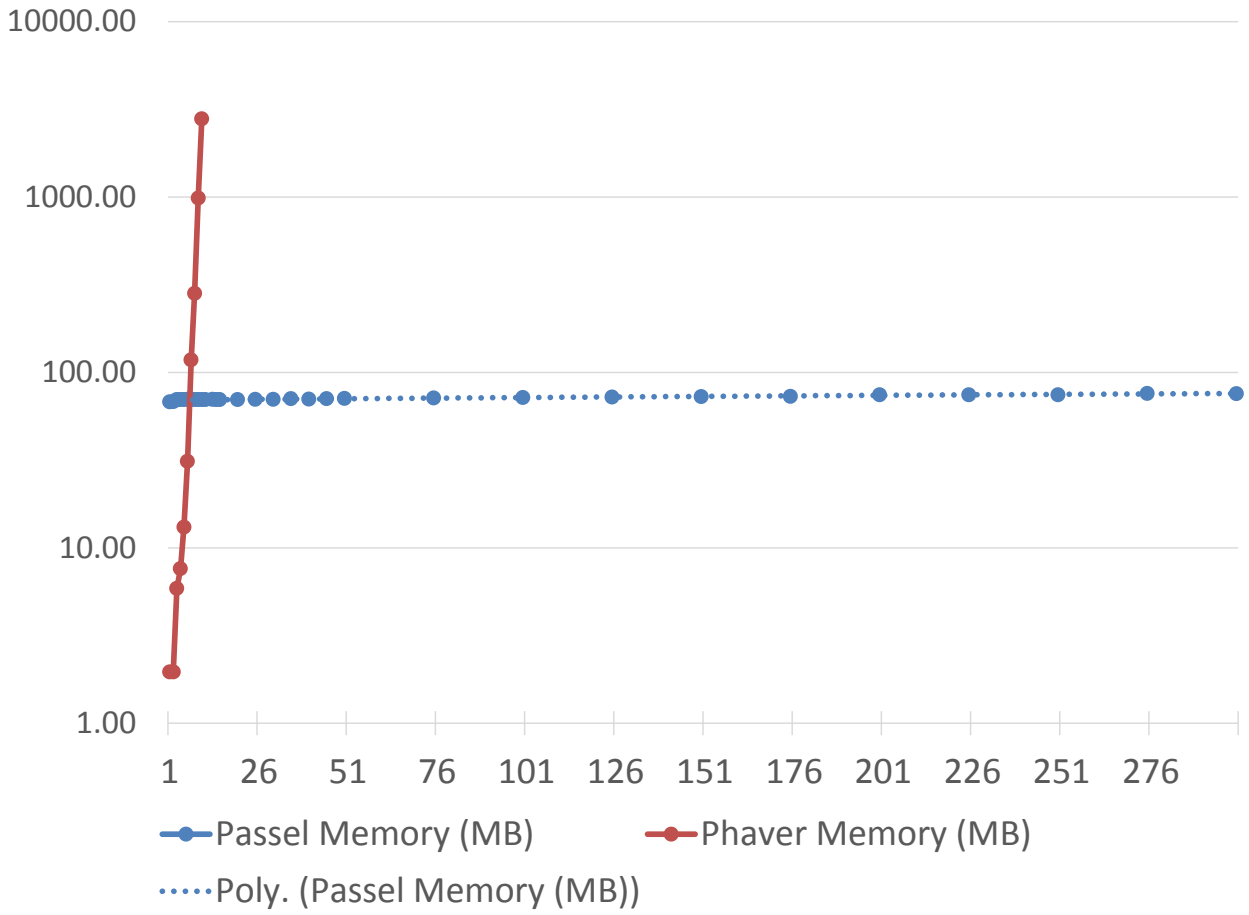

Figure 7.5: Anonymized reachability memory usage comparison of PHAVer and *Passel* for MUX-SEM. The vertical axis scale is logarithmic. The horizontal axis is the number of automata N. This illustrates scaling to hundreds of automata.

MUX-SEM-RA **Mutual Exclusion.** The MUX-SEM-RA mutual exclusion example is just like the MUX-SEM purely discrete mutual exclusion example, except it includes a single continuous local variable for each automaton $\mathcal{A}(i)$. This example illustrates the additional memory and runtime requirements between a discrete and hybrid automaton, as the formulas required to represent the continuous state variables are more complex. As we can see from Figure 7.2, at N = 6, PHAVer requires approximately an order of more runtime to compute the reachable states of MUX-SEM compared to MUX-SEM-RA, at slightly less than one second and around eight seconds seconds, respectively. In comparison, for N = 6, *Passel* requires about four seconds to compute the reach set of MUX-SEM and around eight seconds to compute the reach set of MUX-SEM-RA, a growth of a factor of two, compared to PHAVer's order of magnitude growth. The memory comparison in Figure 7.3 is even better

for *Passel*, even at $\mathsf{N} = 30$ is using under 100 MB memory for MUX-SEM-RA, while PHAVer ran out of memory at $\mathsf{N} = 9$, where it used about 700 MB memory.

SSATS Simplified Small Aircraft Transportation System. The SSATS simplified Small Aircraft Transportation System example is the most challenging example we evaluate. PHAVer was able to compute the reachable states of SSATS up to $\mathsf{N} = 5$ using around 350 MB (as shown in Figure 7.3) and requiring a few minutes (Figure 7.2). In comparison, *Passel* was able to compute the anonymized reachable states of SSATS for up to $\mathsf{N} = 20$ using only about 200 MB memory, although this took about 2.5 hours to complete.

Summary. In summary, comparing all the examples, the anonymized reachability method of Chapter 4 implemented in *Passel* allow us to compute the reachable states of networks composed of many more automata than PHAVer. The real advantage is the memory growth, where even for networks of tens and hundreds of automata, *Passel* never uses more than a few hundred megabytes of memory as shown in Figures 7.3 and 7.5. The runtime required by *Passel* could be reduced by performing some operations more efficiently in the implementation—particularly the checks to determine if a new anonymized state representation is actually new or not—which we plan to implement for future work.

7.7 Experimental Results for Proving Inductive Invariants

This section presents experimental results in using *Passel* to automatically prove the inductive invariance conditions (Definition 2.4) using the methodology of Chapter 5. Table 7.1 summarizes the runtime of *Passel* in automatically checking inductive invariants for several examples. These results summarize *Passel*'s performance when checking a given proof of an inductive invariant $\Gamma(\mathcal{N})$ that implies some desired safety property $\zeta(\mathcal{N})$. This in contrast to Table 7.2, where the inductive invariance checks require more time, since in this case, *Passel* is checking many candidate inductive invariants, some of which are unnecessary in establishing a proof of safety.

As the results indicate, when *Passel* is used to prove that given candidate inductive invari-

Table 7.1: Experimental results for proving inductive invariants. Example properties and results for $2 \leq \mathsf{N} \leq 100$, which exceeds the threshold from Theorem 5.2 for each property and example. SATS properties are shown in Section 2.3.8 and Fischer properties are shown in Section 7.4.1. A check in the "Correct" column means that a property was shown to be an inductive invariant, while an "✗" indicates not, and similarly for buggy versions of the protocols as indicated in the "Buggy" column. Times are the runtimes in seconds. QI is the number of quantifier instantiations.

Example	Property	Correct	Time	QI	Buggy	Time	QI
SSATS	(SA)	✓	0.47	166	✓	24.429	63
	(SB)	✓	0.591	373	✓	0.595	197
	(SC)	✓	0.586	703	✗	1.041	113485
	(SD)	✓	0.757	8298	✗	1.256	1659
SSATS-Timed	(SA)	✓	0.349	66	✓	0.34	61
	(SB)	✓	0.304	673	✓	0.317	460
	(SC)	✓	0.244	373	✗	1.763	140512
	(SE)	✓	0.467	3032	✗	2.204	26958
Fischer-Aux-Timed	(FA)	✓	0.498	305	✓	0.491	305
	(FB)	✓	0.33	204	✓	0.325	204
	(FC)	✓	0.376	544	✓	0.33	544
	(FD)	✓	0.396	618	✗	0.533	2548
	(FE)	✓	0.435	1306	✗	0.532	1202
	(FF)	✓	0.414	1036	✗	0.437	3162

ants are actually inductive invariants, and are sufficient to prove a desired safety property, *Passel* performs efficiently. As mentioned in Section 7.3, *Passel* uses Z3's methods for handling quantified assertions, particularly MBQI and quantifier elimination. The QI columns in Table 7.1 present the number of quantifiers instantiated by the MBQI module. The downside is that these results represent a partially manual effort: the candidate inductive invariants must be specified. Next, we present results on fully automating this process by also finding candidate invariants in Section 7.8.

7.8 Experimental Results for Finding Inductive Invariants

This section presents experimental results using the invariant synthesis methods described in Chapter 6. The invariant synthesis methods are evaluated on several timed and hybrid examples, with the summary of results shown in Table 7.2. We also evaluated a variety of correct and buggy versions of sanity-check protocols (like the purely discrete MUX-SEM example from [41], Fischer's mutual exclusion protocol Fischer, and other protocols we have previously verified). We use a choice of $\mathsf{N} = 3$ for the reachable set computations $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$,

Table 7.2: Experimental invariant synthesis results. All time units are seconds and memory units are megabytes. Checks (✓) in columns $\forall i : \psi(i)?$ and $\dot{\forall} i, j : \psi(i,j)?$ mean that the single and doubly quantified synthesized invariants $\theta(\mathcal{N})$ are inductive, and ✗ means not. Checks (✓) in column $\zeta(\mathcal{N})?$ means *Passel* succeeded in generating a quantified strengthening that implied the desired safety property for all $\mathcal{N} \in \mathbb{N}$ (and ✗, not). Column synth time reports the runtime to synthesize candidate invariants $\forall i : \psi(i)$ and $\dot{\forall} i, j : \psi(i,j)$, and column inv time reports the runtime to prove the candidate assertions are inductive invariants for any $\mathcal{N}$, and that $\forall i : \psi(i) \wedge \dot{\forall} i, j : \psi(i,j) \Rightarrow \zeta(\mathcal{N})$.

	PHAVer		*Passel*				
Name	time	mem	$\forall i : \psi(i)?$	$\dot{\forall} i, j : \psi(i,j)?$	$\zeta(\mathcal{N})?$	synth time	inv time
RFischer_1	1.81	9.94	✓	✓	✓	9.69	138.98
RFischer (Bug)_2	9.88	12.62	✓	✓	✗	34.30	6578.32
TFischer_3	1.31	9.43	✓	✓	✓	8.52	64.95
TFischer (Bug)_4	2.12	12.52	✓	✓	✗	19.59	4169.07
Simple SATS_5	1.72	10.61	✓	✓	✓	4.54	14.70
Sided SATS_6	7.47	22.98	✓	✓	✓	13.95	81.10
TMux_7	0.71	6.74	✓	✓	✓	3.19	20.01
MUX-SEM_8	0.2	5.61	✓	✓	✓	1.68	1.39
MUX-INDEX_9	0.2	6.39	✓	✓	✓	1.38	1.51

but did not see any benefit (in terms of being able to prove inductive properties for more examples) to using greater values in our experiments. We reiterate that when *Passel* proves the inductive invariance conditions, the only assumption on $\mathcal{N}$ is $\mathcal{N} \geq 2$, unless we have a small model theorem with bound N_0—such as Theorem 5.2—and then *Passel* can assume $1 \leq \mathcal{N} \leq \mathsf{N}_0$.

We analyze correct and buggy versions of Fischer's mutual exclusion algorithm Benchmarks 1, 2, 3, and 4, as presented earlier in Figure 2.1. Benchmarks 1 and 2 have rectangular dynamics, so $lb < ub$ and $\dot{x}[i] \in [lb, ub]$, while Benchmarks 3 and 4 have timed dynamics, so $lb = ub$ and $\dot{x}[i] = lb = ub$, which results in fewer synthesized candidate invariants (since there are fewer possible states in the reach set). The timed buggy version has $A \geq B$, while the correct version requires $A < B$. We use numerical values $A = 5$ and $B = 7$ for the correct timed version, and $A = 5$ and $B = 4$ for the buggy version, and $lb = ub = 1$ for the dynamics of both timed cases. While the invariant synthesis procedure requires numerical values (since PHAVer uses numerical libraries for computing the reach set), *Passel* can use

symbolic values (e.g., just an assertion $A < B$ is required for the correct version). The rectangular buggy version has $A = 5$ and $B = 6$, while the correct version uses $A = 5$ and $B = 50$, and $lb = 3$ and $ub = 7$ for the dynamics of both cases. With timed dynamics, Fischer maintains mutual exclusion for $A < B$, but with rectangular dynamics, mutual exclusion requires $B > A * ub$. *Passel* automatically proves mutual exclusion of the correct timed and rectangular versions for any $\mathcal{N} \in \mathbb{N}$ as shown in Benchmarks 1 and 3. The key invariant *Passel* synthesizes for Fischer is related to timing, and for the timed version is:

$$\dot{\forall} i, j \in [\mathcal{N}] : (q[i] = \mathsf{wait} \wedge q[j] = \mathsf{try} \wedge g = i) \Rightarrow (B - A) > (x[i] - x[j]).$$

The buggy versions (Benchmarks 2 and 4) are included as sanity checks, and *Passel* could not prove mutual exclusion of these since it does not hold, although *Passel* did synthesize many inductive invariants. The runtimes of buggy versions are large, because *Passel* attempted to prove many candidate invariants. For example in the timed buggy version, 460 candidate invariants were synthesized and *Passel* proved 98 of them, although these failed to imply mutual exclusion since it is not an invariant.

Passel also proves safety of Benchmarks 5 to 6, which model two simplified versions of the Small Aircraft Transportation System (SATS) [26], where the safety property ζ is that no two aircraft collide. We previously performed a manual deductive strengthening proof for this protocol in [3]—reprinted here as Equation 2.6—and automatically checked it using *Passel* (see Table 7.1). We also analyzed the protocol using the backward reachability technique of Chapter 3 in [2] and here in Section 3.2. Collision avoidance is, for any two aircraft approaching a runway, there is at least a positive real distance L_S between their positions along a one-dimensional line (the path to the runway):

$$\dot{\forall} i, j \in [\mathcal{N}] : (q[i] \in \{\mathbf{b^r}, \mathbf{b^l}, \mathbf{fin}\} \wedge q[j] \in \{\mathbf{b^r}, \mathbf{b^l}, \mathbf{fin}\} \wedge x[i] > x[j]) \Rightarrow x[i] - x[j] \geq L_S.$$

The locations B_R, B_L, etc. specify the aircraft are attempting to land, and $x[i] > x[j]$ ensures aircraft i is ahead of j. The aircraft travel along the line with velocities $\dot{x}[i] \in [v_{min}, v_{max}]$,

and the key invariant synthesized used to establish collision avoidance is

$$\dot{\forall} i, j \in [\mathcal{N}] : (q[i] \in \{\mathbf{b^r}, \mathbf{b^l}, \mathbf{fin}\} \wedge \ q[j] \in \{\mathbf{b^r}, \mathbf{b^l}, \mathbf{fin}\}) \Rightarrow x[i] \geq \frac{(L_B + L_F - x[j])}{(v_{min}(v_{max} - v_{min}))},$$

where L_B and L_F are the lengths of different paths on the way to the runway. Synthesizing this invariant, and others, allowed *Passel* to prove the collision avoidance property fully automatically.

7.9 Summary

In this chapter, we present and evaluate the *Passel* verification tool, which has been successful at automatically proving safety properties of parameterized networks of hybrid automata with rectangular dynamics.[6] *Passel* implements three approaches for verifying safety properties. The first is the the reachability algorithm using anonymized states from Chapter 4. As shown in Section 7.6, this approach has shown very promising results for several examples, such as computing the reachable states for hybrid networks composed of hundreds of automata. While this method cannot solve the general uniform verification problem for parameterized networks since it works with compositions of a finite number N of automata, it can be used in the invariant synthesis procedures described in Chapter 6, and serves as a useful first-pass verification attempt that is easier prior to attempting uniform verification. Additionally, the method is useful for finding counterexamples to show that properties *do not* hold regardless of the choice of $\mathcal{N}$, since, if for a particular N, a desired property does not hold, then it of course cannot hold for all $\mathcal{N} \in \mathbb{N}$.

The second approach *Passel* implements is proving inductive invariance conditions automatically, as described in Chapter 5. As shown in Section 7.7, *Passel* is able to prove the inductive invariance conditions efficiently. The third approach *Passel* implements is finding inductive invariants, as described in Chapter 6. The method is an extension of the invisible invariants method used in a fixed-point procedure for automatically synthesizing inductive invariants. *Passel* then checks these synthesized candidates using the method described

[6] *Passel* and examples are available: https://publish.illinois.edu/passel-tool/.

in Chapter 5. As shown in Section 7.8, *Passel* has performed the first fully automatic verification of several parameterized networks of hybrid automata, such as Fischer's mutual exclusion with rectangular dynamics, part of a conceptual air-traffic control protocol, and several others.

Chapter 8

Conclusion

This dissertation presents the formal modeling and methods for analyzing parameterized networks of hybrid automata, and this chapter concludes the dissertation with a brief summary and directions for future research.

8.1 Summary

This dissertation presents a framework for formally modeling and analyzing safety properties for parameterized networks of hybrid automata. For a more detailed summary of the dissertation and its contributions, refer back to Section 1.2. Chapter 2 presents a new modeling framework for parameterized networks of hybrid automata, along with the input syntax for the *Passel* verification tool developed as a part of this dissertation. The framework allows automata to communicate through a finite number of pointer-like discrete variables, but not through continuous signals. This makes the semantics of composed networks cleaner, but prevents modeling components that can either read continuous signals or signals from arbitrarily many ports. In Chapter 3, we present our results using a backward reachability method and tool MCMT for verifying safety properties of parameterized networks of the subclass of timed automata. We apply this method to an air traffic control case study that motivates our work in subsequent chapters, as the framework allowed in MCMT does not support the generality allowed by our modeling framework of Chapter 2.

The anonymized reachability method of Chapter 4 implemented in *Passel* has yielded promising results, such as being able to compute the reachable states for parameterized networks composed of hundreds of automata. As shown in Chapter 7, *Passel* has yielded promising experimental results and enabled the first automatic verification of parameterized

networks of hybrid automata for several case studies—such as Fischer's mutual exclusion protocol with rectangular dynamics, a simplified version of the Small Aircraft Transportation System (SATS), and several other examples—using the methods of Chapters 5 and 6.

8.2 Future Work

This section describes future research directions based on the results of this dissertation.

8.2.1 Modeling and Specification Extensions

One limitation of the modeling framework developed in Chapter 2 is that it restricts continuous communication between automata, unlike for instance, hybrid input/output automata [9]. This prevents the current framework from being used to specify systems where the dynamics of one automaton depend directly upon the state of another. Generalizing the modeling framework to allow for continuous communication would allow for modeling and verifying systems where the dynamics of one automaton directly depends upon the state of another, such as in standard benchmarks like the heater benchmark [176].

Another direction is to expand the class of properties from safety properties to stability or liveness properties. Given that there is some existing work on automating liveness using invisible invariants like approaches, this may be the most direct avenue for extension [96–98, 160]. The class of liveness properties considered in these works are eventuality properties, for instance, eventually an aircraft lands in SATS. These properties require automatically finding ranking functions that decrease regardless of the number of participants, which is challenging and often requires human ingenuity. Stability properties often need the control-theoretic dual of ranking functions—Lyapunov functions—that also often require human intervention to find. However, there are standard techniques for computing Lyapunov functions for certain classes of dynamics, and there is much active work in developing techniques for applicable to hybrid systems, so developing these techniques for the parameterized setting would be an exciting direction.

8.2.2 *Passel* Extensions

There are many enhancements and extensions that could be made for *Passel* that build upon the extension to its theoretical basis. For instance, *Passel* currently supports analyzing systems with timed ($\dot{x} = 1$) or rectangular ($\dot{x} \in [a, b]$) dynamics, so allowing more general continuous dynamics—such as linear ($\dot{x} = Ax$) or nonlinear ($\dot{x} = f(x)$, for a sufficiently smooth function f) differential equations or inclusions—would expand the class of systems *Passel* can analyze. While allowing for general linear or nonlinear differential equations would likely be intractable—due to decidability results and that the solutions are in general transcendental—some subclasses, such as the class of linear systems with polynomial solutions, may be integrated. The first class of more general dynamics would the subclass of linear systems with polynomials solutions, which can more easily be encoded in SMT solvers than the more general transcendental solutions [177]. From the implementation perspective, Z3 has a new solver for real nonlinear arithmetic, although it is not fully integrated with the other solvers, so in the future this may become a tractable approach [178, 179]. An alternative would be to utilize the SAT modulo ODE approaches [156–158], or the way theorem provers like PVS and MetiTarski handle transcendentals and sinusoids by utilizing truncated Taylor series expansions [180, 181].

8.2.3 Effective Abstractions

One of the most interesting but challenging directions for uniform verification of parameterized networks of hybrid automata is in developing effective abstraction techniques. For instance, network invariants (refer to Sections 1.3.2 and 1.3.3 for an overview) are developed for parameterized networks of timed automata in [49]. We originally attempted extending environment abstraction [80] (see Section 1.3.2 for an overview) to timed systems, but ran into several challenges along the way, such as how to determine the predicates to use in the abstraction. Abstraction techniques developed for hybrid systems such as [182] may be applicable and could lead to automatic uniform verification of more complex hybrid systems than what can be accomplished using the techniques developed in this dissertation.

8.2.4 Applications Requiring Additional Modeling Features

We have analyzed numerous case studies (purely discrete, timed, and rectangular hybrid) in *Passel*. For additional timed protocols, the wireless clock synchronization protocol [183] would serve as a nice example. The Lynch-Shavit timed mutual exclusion algorithm [184] has a large discrete state-space, and would push the limits of the discrete-state analysis in the tool. The fault-tolerant Draper clock synchronization algorithm could be analyzed [185]. There are several other potential timed case studies [186–188]. There are many other CPS case studies that could be formalized and modeled, and we are currently working to find several more examples, potentially some of the protocols from [189–194]. We could also formulate our previous satellite collision avoidance case study as a parameterized problem (we only analyzed it for the case of two satellites previously) [195]. Some of these examples may need to be modeled using linear ($\dot{x}[i] = Ax$) or nonlinear dynamics ($\dot{x}[i] = f(x[i])$), which would also require theoretical extensions and may be interesting to investigate. For example, the subclass of systems with polynomial vector fields could be integrated [177].

Being able to automatically verify properties in the StarL robot platform we have helped develop [196] (or the robots on partitions [17] or robot flocking [16]) would be the highest goal, but many challenges have to be overcome to develop and analyze reasonable models of these system. The classes of systems studied in this dissertation were inspired by our earlier case studies on DCPS [16, 17]. In [17], we analyzed a distributed robotics system in the partitioned plane, where all robots in a particular partition are coupled and move identically, and the software controlling any partition could fail permanently by crashing. We analyzed a distributed flocking protocol in [16], where the actuators of agents could fail by becoming stuck, causing agents to either remain stationary forever, or move in some direction forever. We have not been fully able to analyze these examples automatically, but we have made inroads to their eventual solutions. The current limitations are that these examples involve more complex dynamic communications topologies and linear dynamics. While we could overapproximate the linear dynamics using rectangular differential inclusions, the resulting abstraction was either too coarse to be useful, or too complex to be analyzed.

Bibliography

[1] T. T. Johnson, S. Mitra, and C. Langbort, "Stability of digitally interconnected linear systems," in *Proceedings of the 50th IEEE Conference on Decision and Control and European Control Conference (CDC ECC 2011)*, Orlando, Florida, USA, Dec. 2011, pp. 2687–2692.

[2] T. T. Johnson and S. Mitra, "Parameterized verification of distributed cyber-physical systems: An aircraft landing protocol case study," in *ACM/IEEE 3rd International Conference on Cyber-Physical Systems*, Apr. 2012.

[3] T. T. Johnson and S. Mitra, "A small model theorem for rectangular hybrid automata networks," in *Proceedings of the IFIP International Conference on Formal Techniques for Distributed Systems, Joint 14th Formal Methods for Open Object-Based Distributed Systems and 32nd Formal Techniques for Networked and Distributed Systems (FMOODS-FORTE)*, ser. LNCS. Springer, June 2012, vol. 7273.

[4] E. Frazzoli, M. A. Dahleh, and E. Feron, "Real-time motion planning for agile autonomous vehicles," *AIAA Journal of Guidance, Control, and Dynamics*, vol. 25, no. 1, pp. 116–129, 2002.

[5] A. B. Bosse, W. J. Barnds, M. A. Brown, N. G. Creamer, A. Feerst, C. G. Henshaw, A. S. Hope, B. E. Kelm, P. A. Klein, F. Pipitone, B. E. Plourde, and B. P. Whalen, "SUMO: Spacecraft for the universal modification of orbits," in *Proc. of SPIE*, vol. 5419, 2004, pp. 36–46.

[6] D. Malan, T. Fulford-Jones, M. Welsh, and S. Moulton, "CodeBlue: An ad hoc sensor network infrastructure for emergency medical care," in *Mobisys 2004 Workshop on Applications of Mobile Embedded Systems (WAMES '04)*, J.-P. Hubaux and M. Srivastava, Eds., Boston, MA, 2004.

[7] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. N. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer, M. Gittleman, S. Harbaugh, M. Hebert, T. M. Howard, S. Kolski, A. Kelly, M. Likhachev, M. McNaughton, N. Miller, K. Peterson, B. Pilnick, R. Rajkumar, P. Rybski, B. Salesky, Y.-W. Seo, S. Singh, J. Snider, A. Stentz, W. R. Whittaker, Z. Wolkowicki, J. Ziglar, H. Bae, T. Brown, D. Demitrish, B. Litkouhi, J. Nickolaou, V. Sadekar, W. Zhang, J. Struble, M. Taylor, M. Darms, and D. Ferguson, "Autonomous driving in urban environments: Boss and the urban challenge," *Journal of Field Robotics*, vol. 25, no. 8, pp. 425–466, 2008.

[8] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho, "Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems," in *Hybrid Systems*, R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, Eds. London, UK: Springer-Verlag, 1993, pp. 209–229.

[9] N. Lynch, R. Segala, and F. Vaandrager, "Hybrid I/O automata," *Inf. Comput.*, vol. 185, no. 1, pp. 105–157, 2003.

[10] *Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control (HSCC '11)*. New York, NY, USA: ACM, 2011.

[11] *Proceedings of the 15th ACM International Conference on Hybrid Systems: Computation and Control (HSCC '12)*. New York, NY, USA: ACM, 2012.

[12] C. Belta and F. Ivancic, Eds., *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control (HSCC '13)*. ACM, 2013.

[13] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[14] S. M. Loos, A. Platzer, and L. Nistor, "Adaptive cruise control: Hybrid, distributed, and now formally verified," in *Formal Methods*, ser. LNCS, M. Butler and W. Schulte, Eds. Springer, 2011.

[15] S. Gilbert, N. Lynch, S. Mitra, and T. Nolte, "Self-stabilizing robot formations over unreliable networks," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, pp. 1–17, July 2009.

[16] T. T. Johnson and S. Mitra, "Safe flocking in spite of actuator faults using directional failure detectors," *Journal of Nonlinear Systems and Applications*, vol. 2, no. 1-2, pp. 73–95, Apr. 2011.

[17] T. T. Johnson, S. Mitra, and K. Manamcheri, "Safe and stabilizing distributed cellular flows," in *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems (ICDCS)*. Genoa, Italy: IEEE, June 2010.

[18] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Commun. ACM*, vol. 8, no. 9, p. 569, Sep. 1965.

[19] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, "The algorithmic analysis of hybrid systems," *Theoretical Computer Science*, vol. 138, no. 1, pp. 3–34, 1995.

[20] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, "What's decidable about hybrid automata?" in *ACM Symposium on Theory of Computing*, 1995, pp. 373–382.

[21] D. Liberzon, *Switching in Systems and Control*. Boston, MA, USA: Birkhäuser, 2003.

[22] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager, *The Theory of Timed I/O Automata*, ser. Synthesis Lectures in Computer Science. Morgan & Claypool, 2006.

[23] S. Mitra, "A verification framework for hybrid systems," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA 02139, Sep. 2007.

[24] N. A. Lynch, *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

[25] T. S. Abbott, K. M. Jones, M. C. Consiglio, D. M. Williams, and C. A. Adams, "Small aircraft transportation system, higher volume operations concept: Normal operations," NASA, Tech. Rep. NASA/TM-2004-213022, Aug. 2004.

[26] T. S. Abbott, M. C. Consiglio, B. T. Baxley, D. M. Williams, K. M. Jones, and C. A. Adams, "Small aircraft transportation system higher volume operations concept," NASA, Tech. Rep. NASA/TP-2006-214512, L-19215, Oct. 2006.

[27] C. Muñoz, V. Carreño, and G. Dowek, "Formal analysis of the operational concept for the small aircraft transportation system," in *Rigorous Development of Complex Fault-Tolerant Systems*, ser. LNCS, M. Butler, C. Jones, A. Romanovsky, and E. Troubitsyna, Eds. Springer Berlin / Heidelberg, 2006, vol. 4157, pp. 306–325.

[28] K. R. Apt and D. C. Kozen, "Limits for automatic verification of finite-state concurrent systems," *Inf. Process. Lett.*, vol. 22, no. 6, pp. 307–309, 1986.

[29] I. Suzuki, "Proving properties of a ring of finite-state machines," *Inf. Process. Lett.*, vol. 28, no. 4, pp. 213–214, July 1988.

[30] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, "What's decidable about hybrid automata?" *Journal of Computer and System Sciences*, vol. 57, pp. 94–124, 1998.

[31] G. Lafferriere, G. J. Pappas, and S. Sastry, "O-minimal hybrid systems," *Mathematics of Control, Signals, and Systems (MCSS)*, vol. 13, pp. 1–21, 2000.

[32] A. Carioni, S. Ghilardi, and S. Ranise, "MCMT in the land of parameterized timed automata," in *Proc. of VERIFY 2010*, July 2010.

[33] R. Bruttomesso, A. Carioni, S. Ghilardi, and S. Ranise, "Automated analysis of parametric timing-based mutual exclusion algorithms," in *NASA Formal Methods*, ser. LNCS, A. Goodloe and S. Person, Eds., vol. 7226. Springer Berlin / Heidelberg, 2012, pp. 279–294.

[34] M. Archer, H. Lim, N. Lynch, S. Mitra, and S. Umeno, "Specifying and proving properties of timed I/O automata using Tempo," *Design Automation for Embedded Systems*, vol. 12, pp. 139–170, 2008.

[35] A. Platzer, "Quantified differential dynamic logic for distributed hybrid systems," in *Computer Science Logic*, ser. LNCS, vol. 6247, 2010, pp. 469–483.

[36] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee, "Modular specification of hybrid systems in Charon," in *Hybrid Systems: Computation and Control*, ser. Lecture Notes in Computer Science, N. Lynch and B. Krogh, Eds. Springer Berlin Heidelberg, 2000, vol. 1790, pp. 6–19.

[37] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi, "HyTech: A model checker for hybrid systems," *Journal on Software Tools for Technology Transfer*, vol. 1, pp. 110–122, 1997.

[38] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL: A tool suite for automatic verification of real-time systems," in *Hybrid Systems III*, ser. Lecture Notes in Computer Science, R. Alur, T. Henzinger, and E. Sontag, Eds. Springer Berlin / Heidelberg, 1996, vol. 1066, pp. 232–243.

[39] G. Frehse, "PHAVer: Algorithmic verification of hybrid systems past HyTech," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 10, pp. 263–279, 2008.

[40] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "SpaceEx: Scalable verification of hybrid systems," in *Computer Aided Verification (CAV)*, ser. LNCS. Springer, 2011.

[41] T. Arons, A. Pnueli, S. Ruah, Y. Xu, and L. Zuck, "Parameterized verification with automatically computed inductive assertions?" in *Computer Aided Verification*, ser. LNCS, G. Berry, H. Comon, and A. Finkel, Eds. Springer, 2001, vol. 2102, pp. 221–234.

[42] A. Pnueli, S. Ruah, and L. Zuck, "Automatic deductive verification with invisible invariants," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS. Springer, 2001, vol. 2031, pp. 82–97.

[43] K. Namjoshi, "Symmetry and completeness in the analysis of parameterized systems," in *Verification, Model Checking, and Abstract Interpretation*, ser. Lecture Notes in Computer Science, B. Cook and A. Podelski, Eds. Springer Berlin / Heidelberg, 2007, vol. 4349, pp. 299–313.

[44] S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs I," *Acta Informatica*, vol. 6, pp. 319–340, 1976.

[45] I. Balaban, Y. Fang, A. Pnueli, and L. Zuck, "IIV: An invisible invariant verifier," in *Computer Aided Verification*, ser. LNCS. Springer, 2005, vol. 3576, pp. 293–299.

[46] K. McMillan and L. Zuck, "Invisible invariants and abstract interpretation," in *Static Analysis*, ser. Lecture Notes in Computer Science, E. Yahav, Ed. Springer Berlin / Heidelberg, 2011, vol. 6887, pp. 249–262.

[47] P. A. Abdulla and B. Jonsson, "Model checking of systems with many identical timed processes," *Theoretical Computer Science*, vol. 290, no. 1, pp. 241–264, 2003.

[48] P. A. Abdulla, J. Deneux, and P. Mahata, "Multi-clock timed networks," in *Proc. of 19th Annual IEEE Symposium Logic in Computer Science*, July 2004, pp. 345–354.

[49] O. Grinchtein and M. Leucker, "Network invariants for real-time systems," *Formal Aspects of Computing*, vol. 20, pp. 619–635, 2008.

[50] T. Göthel, "Mechanical verification of parameterized real-time systems," Ph.D. dissertation, Technische Universität Berlin, 2012.

[51] P. Abdulla, G. Delzanno, and A. Rezine, "Parameterized verification of infinite-state processes with global conditions," in *Computer Aided Verification*, ser. LNCS, W. Damm and H. Hermanns, Eds. Springer, 2007, vol. 4590, pp. 145–157.

[52] P. A. Abdulla, N. Henda, G. Delzanno, and A. Rezine, "Handling parameterized systems with non-atomic global conditions," in *Verification, Model Checking, and Abstract Interpretation*, ser. Lecture Notes in Computer Science, F. Logozzo, D. A. Peled, and L. Zuck, Eds. Springer Berlin Heidelberg, 2008, vol. 4905, pp. 22–36.

[53] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli, "Towards SMT model checking of array-based systems," in *Automated Reasoning*, ser. LNCS. Springer, 2008, vol. 5195, pp. 67–82.

[54] S. Ghilardi and S. Ranise, "MCMT: A model checker modulo theories," in *Automated Reasoning*, ser. LNCS. Springer, 2010, vol. 6173, pp. 22–29.

[55] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina, "SAFARI: SMT-based abstraction for arrays with interpolants," in *24th International Conference on Computer Aided Verification (CAV)*, Springer. Berkeley, California, USA: Springer, 2012.

[56] S. Conchon, A. Goel, S. Krstić, A. Mebsout, and F. Zaïdi, "Cubicle: A parallel SMT-based model checker for parameterized systems," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, P. Madhusudan and S. Seshia, Eds. Springer Berlin / Heidelberg, 2012, vol. 7358, pp. 718–724.

[57] A. Platzer and E. Clarke, "Computing differential invariants of hybrid systems as fixedpoints," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, A. Gupta and S. Malik, Eds. Springer Berlin / Heidelberg, 2008, vol. 5123, pp. 176–189.

[58] A. Platzer and E. Clarke, "Formal verification of curved flight collision avoidance maneuvers: A case study," in *Formal Methods*, ser. LNCS, A. Cavalcanti and D. Dams, Eds. Springer, 2009, vol. 5850, pp. 547–562.

[59] N. Lynch, "Modelling and verification of automated transit systems, using timed automata, invariants and simulations," in *Proceedings of the DIMACS/SYCON Workshop on Hybrid Systems III: Verification and Control*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1996, pp. 449–463.

[60] E. Dolginova and N. Lynch, "Safety verification for automated platoon maneuvers: A case study," in *HART '97 (International Workshop on Hybrid and Real-Time Systems)*, ser. LNCS, vol. 1201. Springer Verlag, March 1997.

[61] C. Livadas, J. Lygeros, and N. A. Lynch, "High-level modeling and analysis of TCAS," in *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS '99)*, Dec. 1999, pp. 115–125.

[62] C. Tomlin, G. Pappas, and S. Sastry, "Conflict resolution for air traffic management: A study in multiagent hybrid systems," *IEEE Trans. Autom. Control*, vol. 43, no. 4, pp. 509–521, Apr. 1998.

[63] C. Tomlin, I. Mitchell, and R. Ghosh, "Safety verification of conflict resolution maneuvers," *Intelligent Transportation Systems, IEEE Transactions on*, vol. 2, no. 2, pp. 110–120, June 2001.

[64] A. Bayen, I. M. Mitchell, M. M. K. Oishi, and C. J. Tomlin, "Aircraft autolander safety analysis through optimal control-based reach set computation," *Journal of Guidance, Control, and Dynamics*, vol. 30, no. 1, Jan. 2007.

[65] C. Muñoz, G. Dowek, and V. Carreño, "Modeling and verification of an air traffic concept of operations," *Software Engineering Notes*, vol. 29, no. 4, pp. 175–182, 2004.

[66] C. Muñoz and G. Dowek, "Hybrid verification of an air traffic operational concept," in *Proceedings of IEEE ISoLA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation*, Columbia, Maryland, 2005.

[67] V. Carreño and C. Muñoz, "Safety verification of the small aircraft transportation system concept of operations," in *Proceedings of the AIAA 5th Aviation, Technology, Integration, and Operations Conference, AIAA-2005-7423*, Arlington, Virginia, 2005.

[68] S. Umeno and N. Lynch, "Proving safety properties of an aircraft landing protocol using I/O automata and the PVS theorem prover: A case study," in *Formal Methods*, ser. LNCS, J. Misra, T. Nipkow, and E. Sekerinski, Eds. Springer, 2006, vol. 4085, pp. 64–80.

[69] S. Umeno and N. Lynch, "Safety verification of an aircraft landing protocol: A refinement approach," in *Hybrid Systems: Computation and Control*, ser. LNCS. Springer, 2007, vol. 4416, pp. 557–572.

[70] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A prototype verification system," in *11th International Conference on Automated Deduction (CADE)*, ser. LNAI, D. Kapur, Ed., vol. 607. Saratoga, NY: Springer-Verlag, June 1992, pp. 748–752.

[71] P. Prabhakar, P. S. Duggirala, S. Mitra, and M. Viswanathan, "Hybrid automata-based CEGAR for rectangular hybrid systems," in *VMCAI*, 2013, pp. 48–67.

[72] L. Zuck and A. Pnueli, "Model checking and abstraction to the aid of parameterized systems," *Computer Languages, Systems, and Structures*, vol. 30, no. 3-4, pp. 139–169, 2004.

[73] B. D. Lubachevsky, "An approach to automating the verification of compact parallel coordination programs," *Acta Informatica*, vol. 21, no. 2, pp. 125–169, 1984.

[74] E. M. Clarke, O. Grumberg, and M. C. Browne, "Reasoning about networks with many identical finite-state processes," in *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing (PODC '86)*. New York, NY, USA: ACM, 1986, pp. 240–248.

[75] S. M. German and A. P. Sistla, "Reasoning about systems with many processes," *J. ACM*, vol. 39, no. 3, pp. 675–735, 1992.

[76] G. Delzanno, "Automatic verification of parameterized cache coherence protocols," in *Computer Aided Verification*, ser. LNCS, E. Emerson and A. Sistla, Eds. Springer Berlin / Heidelberg, 2000, vol. 1855, pp. 53–68.

[77] F. Pong and M. Dubois, "A new approach for the verification of cache coherence protocols," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 6, no. 8, pp. 773–787, 1995.

[78] K. McMillan, "Parameterized verification of the FLASH cache coherence protocol by compositional model checking," in *Correct Hardware Design and Verification Methods*, ser. Lecture Notes in Computer Science, T. Margaria and T. Melham, Eds. Springer Berlin / Heidelberg, 2001, vol. 2144, pp. 179–195.

[79] A. Pnueli, J. Xu, and L. Zuck, "Liveness with $(0, 1, \infty)$-counter abstraction," in *Computer Aided Verification*, ser. LNCS, E. Brinksma and K. Larsen, Eds. Springer Berlin / Heidelberg, 2002, vol. 2404, pp. 93–111.

[80] E. Clarke, M. Talupur, and H. Veith, "Environment abstraction for parameterized verification," in *Verification, Model Checking, and Abstract Interpretation*, ser. LNCS, E. Emerson and K. Namjoshi, Eds. Springer, 2006, vol. 3855, pp. 126–141.

[81] P. A. Abdulla, G. Delzanno, N. B. Henda, and A. Rezine, "Monotonic abstraction (on efficient verification of parameterized systems)," *International Journal of Foundations of Computer Science*, vol. 20, no. 5, pp. 779–801, 2009.

[82] P. Abdulla, Y.-F. Chen, G. Delzanno, F. Haziza, C.-D. Hong, and A. Rezine, "Constrained monotonic abstraction: A CEGAR for parameterized verification," in *CONCUR 2010*, ser. Lecture Notes in Computer Science, P. Gastin and F. Laroussinie, Eds. Springer Berlin / Heidelberg, 2010, vol. 6269, pp. 86–101.

[83] P. A. Abdulla, B. Jonsson, M. Nilsson, and M. Saksena, "A survey of regular model checking," in *Concurrency Theory (CONCUR 2004)*, ser. Lecture Notes in Computer Science, P. Gardner and N. Yoshida, Eds. Springer Berlin / Heidelberg, 2004, vol. 3170, pp. 35–48.

[84] A. Legay and P. Wolper, "On (omega-)regular model checking," *ACM Trans. Comput. Logic*, vol. 12, pp. 2:1–2:46, Nov. 2010.

[85] P. Wolper and V. Lovinfosse, "Verifying properties of large sets of processes with network invariants," in *Automatic Verification Methods for Finite State Systems*, ser. LNCS, J. Sifakis, Ed. Springer Berlin / Heidelberg, 1990, vol. 407, pp. 68–80.

[86] F. Balarin and A. Sangiovanni-Vincentelli, "On the automatic computation of network invariants," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, D. Dill, Ed. Springer Berlin / Heidelberg, 1994, vol. 818, pp. 234–246.

[87] E. M. Clarke, O. Grumberg, and S. Jha, "Verifying parameterized networks," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 5, pp. 726–750, Sep. 1997.

[88] P. Abdulla and B. Jonsson, "On the existence of network invariants for verifying parameterized systems," in *Correct System Design*, ser. Lecture Notes in Computer Science, E.-R. Olderog and B. Steffen, Eds. Springer Berlin / Heidelberg, 1999, vol. 1710, pp. 180–197.

[89] D. Lesens, N. Halbwachs, and P. Raymond, "Automatic verification of parameterized networks of processes," *Theoretical Computer Science*, vol. 256, no. 1-2, pp. 113–144, 2001.

[90] Y. Kesten, A. Pnueli, E. Shahar, and L. Zuck, "Network invariants in action," in *CONCUR 2002 ? Concurrency Theory*, ser. Lecture Notes in Computer Science, L. Brim, M. Kretnsk, A. Kucera, and P. Jancar, Eds. Springer Berlin / Heidelberg, 2002, vol. 2421, pp. 217–264.

[91] O. Grinchtein, M. Leucker, and N. Piterman, "Inferring network invariants automatically," in *Automated Reasoning*, ser. Lecture Notes in Computer Science, U. Furbach and N. Shankar, Eds. Springer Berlin / Heidelberg, 2006, vol. 4130, pp. 483–497.

[92] T. Göthel and S. Glesner, "Towards the semi-automatic verification of parameterized real-time systems using network invariants," in *Software Engineering and Formal Methods (SEFM), 2010 8th IEEE International Conference on*, Sep. 2010, pp. 310–314.

[93] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer, 2002, vol. 2283.

[94] T. Göthel and S. Glesner, "An approach for machine-assisted verification of timed CSP specifications," *Innovations in Systems and Software Engineering*, vol. 6, pp. 181–193, 2010.

[95] N. Piterman, "Verification of infinite-state systems," Ph.D. dissertation, Weizmann Institute of Science, 2004.

[96] Y. Fang, N. Piterman, A. Pnueli, and L. Zuck, "Liveness with invisible ranking," in *Verification, Model Checking, and Abstract Interpretation*, ser. Lecture Notes in Computer Science, B. Steffen and G. Levi, Eds. Springer Berlin / Heidelberg, 2004, vol. 2937, pp. 109–132.

[97] Y. Fang, N. Piterman, A. Pnueli, and L. Zuck, "Liveness with invisible ranking," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 8, pp. 261–279, 2006.

[98] Y. Fang, K. McMillan, A. Pnueli, and L. Zuck, "Liveness by invisible invariants," in *Formal Techniques for Networked and Distributed Systems*, ser. Lecture Notes in Computer Science, E. Najm, J.-F. Pradat-Peyre, and V. Donzeau-Gouge, Eds. Springer, 2006, vol. 4229, pp. 356–371.

[99] E. Emerson and V. Kahlon, "Reducing model checking of the many to the few," in *Automated Deduction (CADE-17)*, ser. Lecture Notes in Computer Science, D. McAllester, Ed. Springer Berlin / Heidelberg, 2000, vol. 1831, pp. 236–254.

[100] E. Emerson and V. Kahlon, "Model checking large-scale and parameterized resource allocation systems," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, J.-P. Katoen and P. Stevens, Eds. Springer Berlin / Heidelberg, 2002, vol. 2280, pp. 55–69.

[101] Q. Yang and M. Li, "A cut-off approach for bounded verification of parameterized systems," in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 1, 2010, pp. 345–354.

[102] Y. Hanna, D. Samuelson, S. Basu, and H. Rajan, "Automating cut-off for multi-parameterized systems," in *Formal Methods and Software Engineering*, ser. LNCS, J. Dong and H. Zhu, Eds. Springer Berlin / Heidelberg, 2010, vol. 6447, pp. 338–354.

[103] B. Dutertre and M. Sorea, "Timed systems in SAL," SRI International, Tech. Rep. SRI-SDL-04-03, Oct. 2004.

[104] A. Platzer, "Quantified differential invariants," in *Proc. of the 14th ACM Intl. Conf. on Hybrid Systems: Computation and Control*. ACM, 2011, pp. 63–72.

[105] A. Annichini, A. Bouajjani, and M. Sighireanu, "TReX: A tool for reachability analysis of complex systems," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, G. Berry, H. Comon, and A. Finkel, Eds. Springer Berlin / Heidelberg, 2001, vol. 2102, pp. 368–372.

[106] J. Faber, C. Ihlemann, S. Jacobs, and V. Sofronie-Stokkermans, "Automatic verification of parametric specifications with complex topologies," in *Integrated Formal Methods*, ser. LNCS. Springer, 2010, vol. 6396, pp. 152–167.

[107] L. Lamport, "A fast mutual exclusion algorithm," *ACM Trans. Comput. Syst.*, vol. 5, no. 1, pp. 1–11, 1987.

[108] D. Lesens and H. Saidi, "Automatic verification of parameterized networks of processes by abstraction," *Electronic Notes of Theoretical Computer Science*, 1997.

[109] P. Abdulla, G. Delzanno, O. Rezine, A. Sangnier, and R. Traverso, "On the verification of timed ad hoc networks," in *Formal Modeling and Analysis of Timed Systems*, ser. Lecture Notes in Computer Science, U. Fahrenberg and S. Tripakis, Eds. Springer Berlin / Heidelberg, 2011, vol. 6919, pp. 256–270.

[110] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.

[111] A. Puri and P. Varaiya, "Decidability of hybrid systems with rectangular differential inclusions," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, D. Dill, Ed. Springer Berlin / Heidelberg, 1994, vol. 818, pp. 95–104.

[112] H. Lim, D. Kaynar, N. Lynch, and S. Mitra, "Translating timed I/O automata specifications for theorem proving in PVS," in *Formal Modeling and Analysis of Timed Systems*, ser. LNCS, P. Pettersson and W. Yi, Eds. Springer, 2005, vol. 3829, pp. 17–31.

[113] L. de Moura, S. Owre, H. Ruess, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari, "SAL 2," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, R. Alur and D. A. Peled, Eds. Springer Berlin Heidelberg, 2004, vol. 3114, pp. 496–500.

[114] E. M. Clarke, R. Enders, T. Filkorn, and S. Jha, "Exploiting symmetry in temporal logic model checking," *Formal Methods in System Design*, vol. 9, pp. 77–104, 1996.

[115] C. N. Ip and D. L. Dill, "Better verification through symmetry," *Formal Methods in System Design*, vol. 9, pp. 41–75, 1996.

[116] E. A. Emerson and A. P. Sistla, "Symmetry and model checking," *Formal Methods in System Design*, vol. 9, no. 1-2, pp. 105–131, 1996.

[117] W. D. Obal II, "Measure-adaptive state-space construction methods," Ph.D. dissertation, University of Arizona, 1998.

[118] W. D. Obal and W. H. Sanders, "Measure-adaptive state-space construction," *Performance Evaluation*, vol. 44, no. 1–4, pp. 237–258, 2001.

[119] E. Emerson and T. Wahl, "Dynamic symmetry reduction," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, N. Halbwachs and L. Zuck, Eds. Springer Berlin Heidelberg, 2005, vol. 3440, pp. 382–396.

[120] W. D. Obal, M. McQuinn, and W. Sanders, "Detecting and exploiting symmetry in discrete-state Markov models," *Reliability, IEEE Transactions on*, vol. 56, no. 4, pp. 643–654, Dec. 2007.

[121] T. Wahl, N. Blanc, and E. Emerson, "SVISS: Symbolic verification of symmetric systems," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, C. Ramakrishnan and J. Rehof, Eds. Springer Berlin Heidelberg, 2008, vol. 4963, pp. 459–462.

[122] G. Basler, M. Mazzucchi, T. Wahl, and D. Kroening, "Symbolic counter abstraction for concurrent software," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, A. Bouajjani and O. Maler, Eds. Springer Berlin Heidelberg, 2009, vol. 5643, pp. 64–78.

[123] D. L. Dill, "The murφ verification system," in *Proceedings of the 8th International Conference on Computer Aided Verification*, ser. CAV '96. London, UK, UK: Springer-Verlag, 1996, pp. 390–393.

[124] C. N. Ip and D. L. Dill, "Verifying systems with replicated components in Murφ," *Formal Methods in System Design*, vol. 14, no. 3, May 1999.

[125] M. Hendriks, G. Behrmann, K. G. Larsen, P. Niebert, and F. W. Vaandrager, "Adding symmetry reduction to UPPAAL," in *Formal Modeling and Analysis of Timed Systems (FORMATS '03)*, ser. LNCS, K. G. Larsen and P. Niebert, Eds., no. 2791. Springer–Verlag, 2004, pp. 46–59.

[126] M. Hendriks, "Model checking timed automata: Techniques and applications," Ph.D. dissertation, University of Nijmegen, The Netherlands, 2006.

[127] R. Bryant, "Graph-based algorithms for Boolean function manipulation," *Computers, IEEE Transactions on*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.

[128] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, W. Cleaveland, Ed. Springer Berlin / Heidelberg, 1999, vol. 1579, pp. 193–207.

[129] D. Dill, "Timing assumptions and verification of finite-state concurrent systems," in *Automatic Verification Methods for Finite State Systems*, ser. Lecture Notes in Computer Science, J. Sifakis, Ed. Springer Berlin / Heidelberg, 1990, vol. 407, pp. 197–212.

[130] S. Yovine, "Model checking timed automata," in *Lectures on Embedded Systems*, ser. Lecture Notes in Computer Science, G. Rozenberg and F. Vaandrager, Eds. Springer Berlin / Heidelberg, 1998, vol. 1494, pp. 114–152.

[131] R. Alur, "Timed automata," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, N. Halbwachs and D. Peled, Eds. Springer Berlin / Heidelberg, 1999, vol. 1633, pp. 688–688.

[132] J. Moller, J. Lichtenberg, H. Andersen, and H. Hulgaard, "Difference decision diagrams," in *Computer Science Logic*, ser. Lecture Notes in Computer Science, J. Flum and M. Rodriguez-Artalejo, Eds. Springer Berlin / Heidelberg, 2009, vol. 1683, pp. 826–826.

[133] A. Girard, "Reachability of uncertain linear systems using zonotopes," in *Hybrid Systems: Computation and Control*, ser. Lecture Notes in Computer Science, M. Morari and L. Thiele, Eds. Springer Berlin / Heidelberg, 2005, vol. 3414, pp. 291–305.

[134] M. Althoff, O. Stursberg, and M. Buss, "Computing reachable sets of hybrid systems using a combination of zonotopes and polytopes," *Nonlinear Analysis: Hybrid Systems*, vol. 4, no. 2, pp. 233–249, 2010.

[135] S. Ghilardi and S. Ranise, "Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis," *Logical Methods in Computer Science*, vol. 6, no. 4, 2010.

[136] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. of 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS '08/ETAPS '08. Springer-Verlag, 2008, pp. 337–340.

[137] A. Filippov, "Classical solutions of differential equations with multi-valued right-hand side," *SIAM Journal on Control*, vol. 5, no. 4, pp. 609–621, 1967.

[138] S. Viken and F. Brooks, "Demonstration of four operating capabilities to enable a small aircraft transportation system," in *The 24th Digital Avionics Systems Conference (DASC 2005)*, vol. 2, Oct. 2005.

[139] H. K. Khalil, *Nonlinear Systems*, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2002.

[140] G. Lafferriere, G. Pappas, and S. Yovine, "A new class of decidable hybrid systems," in *Hybrid Systems: Computation and Control*, ser. Lecture Notes in Computer Science, F. Vaandrager and J. van Schuppen, Eds. Springer Berlin / Heidelberg, 1999, vol. 1569, pp. 137–151.

[141] C. L. Guernic and A. Girard, "Reachability analysis of linear systems using support functions," *Nonlinear Analysis: Hybrid Systems*, vol. 4, no. 2, pp. 250–262, 2010.

[142] R. Floyd, "Assigning meanings to programs," *Mathematical Aspects of Computer Science*, vol. 19, no. 19-32, p. 1, 1967.

[143] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969.

[144] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, pp. 453–457, Aug. 1975.

[145] G. Chockler, N. Lynch, S. Mitra, and J. Tauber, "Proving atomicity: An assertional approach," in *Distributed Computing*, ser. Lecture Notes in Computer Science, P. Fraigniaud, Ed. Springer Berlin Heidelberg, 2005, vol. 3724, pp. 152–168.

[146] A. Finkel and P. Schnoebelen, "Well-structured transition systems everywhere!" *Theoretical Computer Science*, vol. 256, no. 1-2, pp. 63–92, 2001.

[147] B. Dutertre and L. De Moura, "The Yices SMT solver," SRI International, Tech. Rep., 2006.

[148] F. Alberti, S. Ghilardi, E. Pagani, S. Ranise, and G. P. Rossi, "Universal guards, relativization of quantifiers, and failure models in model checking modulo theories," *JSAT*, vol. 8, no. 1/2, pp. 29–61, 2012.

[149] H. Mangassarian, A. Veneris, and M. Benedetti, "Robust QBF encodings for sequential circuits with applications to verification, debug, and test," *Computers, IEEE Transactions on*, vol. 59, no. 7, pp. 981–994, July 2010.

[150] L. De Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications," *Commun. ACM*, vol. 54, pp. 69–77, Sep. 2011.

[151] G. Audemard, A. Cimatti, A. Kornilowicz, and R. Sebastiani, "Bounded model checking for timed systems," in *Formal Techniques for Networked and Distributed Systems (FORTE)*, ser. LNCS, D. Peled and M. Vardi, Eds. Springer Berlin / Heidelberg, 2002, vol. 2529, pp. 243–259.

[152] M. Sorea, "Bounded model checking for timed automata," *Electronic Notes in Theoretical Computer Science*, vol. 68, no. 5, pp. 116–134, 2002.

[153] R. Kindermann, T. Junttila, and I. Niemelä, "Beyond lassos: Complete SMT-based bounded model checking for timed automata," in *Formal Techniques for Distributed Systems*, ser. Lecture Notes in Computer Science, H. Giese and G. Rosu, Eds. Springer Berlin Heidelberg, 2012, vol. 7273, pp. 84–100.

[154] R. Kindermann, T. Junttila, and I. Niemelä, "SMT-based induction methods for timed systems," in *Formal Modeling and Analysis of Timed Systems*, ser. Lecture Notes in Computer Science, M. Jurdziski and D. Nickovic, Eds. Springer Berlin Heidelberg, 2012, vol. 7595, pp. 171–187.

[155] A. Cimatti, S. Mover, and S. Tonetta, "SMT-based scenario verification for hybrid systems," *Formal Methods in System Design*, pp. 1–21, 2012.

[156] A. Eggers, M. Fränzle, and C. Herde, "SAT modulo ODE: A direct SAT approach to hybrid systems," in *Automated Technology for Verification and Analysis*, ser. Lecture Notes in Computer Science, S. Cha, J.-Y. Choi, M. Kim, I. Lee, and M. Viswanathan, Eds. Springer Berlin / Heidelberg, 2008, vol. 5311, pp. 171–185.

[157] D. Ishii, K. Ueda, and H. Hosobe, "An interval-based SAT modulo ODE solver for model checking nonlinear hybrid systems," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 13, pp. 449–461, 2011.

[158] A. Eggers, N. Ramdani, N. Nedialkov, and M. Fränzle, "Improving SAT modulo ODE for hybrid systems analysis by combining different enclosure methods," in *Software Engineering and Formal Methods*, ser. Lecture Notes in Computer Science, G. Barthe, A. Pardo, and G. Schneider, Eds. Springer Berlin / Heidelberg, 2011, vol. 7041, pp. 172–187.

[159] E. Börger, E. Grädel, and Y. Gurevich, *The Classical Decision Problem*. Springer, 2001.

[160] Y. Fang, N. Piterman, A. Pnueli, and L. Zuck, "Liveness with incomprehensible ranking," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, K. Jensen and A. Podelski, Eds. Springer, 2004, vol. 2988, pp. 482–496.

[161] Y. Ge and L. de Moura, "Complete instantiation for quantified formulas in satisfiabiliby modulo theories," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, A. Bouajjani and O. Maler, Eds. Springer Berlin / Heidelberg, 2009, vol. 5643, pp. 306–320.

[162] N. Bjørner, "Linear quantifier elimination as an abstract decision procedure," in *Automated Reasoning*, ser. LNCS. Springer, 2010, vol. 6173, pp. 316–330.

[163] G. Brown and L. Pike, "Easy parameterized verification of biphase mark and 8N1 protocols," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS. Springer, 2006, vol. 3920, pp. 58–72.

[164] A. Donaldson, L. Haller, D. Kroening, and P. Rmmer, "Software verification using k-induction," in *Static Analysis*, ser. LNCS, E. Yahav, Ed. Springer Berlin / Heidelberg, 2011, vol. 6887, pp. 351–368.

[165] I. Balaban, A. Pnueli, and L. Zuck, "Invisible safety of distributed protocols," in *Automata, Languages and Programming*, ser. Lecture Notes in Computer Science, M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, Eds. Springer Berlin / Heidelberg, 2006, vol. 4052, pp. 528–539.

[166] A. Pnueli and E. Shahar, "A platform for combining deductive with algorithmic verification," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, R. Alur and T. Henzinger, Eds. Springer Berlin / Heidelberg, 1996, vol. 1102, pp. 184–195.

[167] A. Tarski, *A Decision Method for Elementary Algebra and Geometry*. Santa Monica, CA: RAND Corporation, 1951.

[168] G. Nelson and D. C. Oppen, "Simplification by cooperating decision procedures," *ACM Trans. Program. Lang. Syst.*, vol. 1, no. 2, pp. 245–257, Oct. 1979.

[169] R. E. Shostak, "Deciding combinations of theories," in *6th Conference on Automated Deduction*, ser. Lecture Notes in Computer Science, D. Loveland, Ed. Springer Berlin Heidelberg, 1982, vol. 138, pp. 209–222.

[170] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "DPLL(T): Fast decision procedures," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, R. Alur and D. Peled, Eds. Springer Berlin / Heidelberg, 2004, vol. 3114, pp. 293–295.

[171] S. C. Kleene, *Introduction to metamathematics*. North Holland, 1980.

[172] A. Cohen and K. Namjoshi, "Local proofs for global safety properties," *Formal Methods in System Design*, vol. 34, pp. 104–125, 2009.

[173] K. Manamcheri, S. Mitra, S. Bak, and M. Caccamo, "A step towards verification and synthesis from Simulink/Stateflow models," in *Proc. of the 14th Intl. Conf. on Hybrid Systems: Computation and Control (HSCC)*. ACM, 2011, pp. 317–318.

[174] K. Hoder, N. Bjørner, and L. de Moura, "muZ – An efficient engine for fixed points with constraints," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds. Springer Berlin / Heidelberg, 2011, vol. 6806, pp. 457–462.

[175] K. Hoder and N. Bjørner, "Generalized property directed reachability," in *Theory and Applications of Satisfiability Testing (SAT)*, ser. Lecture Notes in Computer Science, A. Cimatti and R. Sebastiani, Eds. Springer Berlin / Heidelberg, 2012, vol. 7317, pp. 157–171.

[176] A. Fehnker and F. Ivancic, "Benchmarks for hybrid systems verification," in *Hybrid Systems: Computation and Control (HSCC '04)*, ser. Lecture Notes in Computer Science, R. Alur and G. J. Pappas, Eds. Springer Berlin Heidelberg, 2004, vol. 2993, pp. 326–341.

[177] G. Lafferriere, G. J. Pappas, and S. Yovine, "Symbolic reachability computation for families of linear vector fields," *J. Symb. Comput.*, vol. 32, no. 3, pp. 231–253, 2001.

[178] D. Jovanović and L. de Moura, "Solving non-linear arithmetic," in *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK. Proceedings*, ser. Lecture Notes in Computer Science, vol. 7364. Springer, June 2012, pp. 339–354.

[179] L. de Moura and G. Passmore, "Computation in real closed infinitesimal and transcendental extensions of the rationals," in *Proceedings of 24th International Conference on Automated Deduction (CADE-24)*, June 2013.

[180] M. Daumas, D. Lester, and C. Muñoz, "Verified real number calculations: A library for interval arithmetic," *Computers, IEEE Transactions on*, vol. 58, no. 2, pp. 226–237, Feb. 2009.

[181] L. Paulson, "MetiTarski: Past and future," in *Interactive Theorem Proving*, ser. Lecture Notes in Computer Science, L. Beringer and A. Felty, Eds. Springer Berlin / Heidelberg, 2012, vol. 7406, pp. 1–10.

[182] R. Alur, T. Henzinger, G. Lafferriere, and G. Pappas, "Discrete abstractions of hybrid systems," *Proceedings of the IEEE*, vol. 88, no. 7, pp. 971–984, July 2000.

[183] F. Heidarian, J. Schmaltz, and F. Vaandrager, "Analysis of a clock synchronization protocol for wireless sensor networks," *Theoretical Computer Science*, vol. 413, no. 1, pp. 87–105, 2012.

[184] N. Lynch and N. Shavit, "Timing-based mutual exclusion," in *Real-Time Systems Symposium, 1992*, Dec. 1992, pp. 2–11.

[185] W. Daly, J. Hopkins, A.L., and J. McKenna, "A fault-tolerant digital clocking system," in *Twenty-Fifth International Symposium on Fault-Tolerant Computing (FTCS '95), Highlights from Twenty-Five Years*, 1973, p. 419.

[186] W. Steiner, J. Rushby, M. Sorea, and H. Pfeifer, "Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation," in *International Conference on Dependable Systems and Networks (DSN '04)*, June 2004, pp. 189–198.

[187] G. Rodriguez-Navas, J. Proenza, and H. Hansson, "Using UPPAAL to model and verify a clock synchronization protocol for the controller area network," in *10th IEEE Conference on Emerging Technologies and Factory Automation (ETFA '05)*, vol. 2, Sep. 2005, p. 8.

[188] B. Dutertre and M. Sorea, "Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata," in *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, ser. Lecture Notes in Computer Science, Y. Lakhnech and S. Yovine, Eds. Springer Berlin / Heidelberg, 2004, vol. 3253, pp. 1–6.

[189] J. Heo, D. Henriksson, X. Liu, and T. Abdelzaher, "Integrating adaptive components: An emerging challenge in performance-adaptive systems and a server farm case-study," in *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, Dec. 2007, pp. 227–238.

[190] M. Khan, L. Luo, C. Huang, and T. Abdelzaher, "SNTS: Sensor network troubleshooting suite," in *Distributed Computing in Sensor Systems*, ser. Lecture Notes in Computer Science, J. Aspnes, C. Scheideler, A. Arora, and S. Madden, Eds. Springer Berlin / Heidelberg, 2007, vol. 4549, pp. 142–157.

[191] M. M. H. Khan, H. K. Le, H. Ahmadi, T. F. Abdelzaher, and J. Han, "Dustminer: Troubleshooting interactive complexity bugs in sensor networks," in *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems (SenSys)*. New York, NY, USA: ACM, 2008, pp. 99–112.

[192] M. M. H. Khan, T. Abdelzaher, and K. Gupta, "Towards diagnostic simulation in sensor networks," in *Distributed Computing in Sensor Systems*, ser. Lecture Notes in Computer Science, S. Nikoletseas, B. Chlebus, D. Johnson, and B. Krishnamachari, Eds. Springer Berlin / Heidelberg, 2008, vol. 5067, pp. 252–265.

[193] M. M. H. Khan, T. Abdelzaher, J. Han, and H. Ahmadi, "Finding symbolic bug patterns in sensor networks," in *Distributed Computing in Sensor Systems*, ser. Lecture Notes in Computer Science, B. Krishnamachari, S. Suri, W. Heinzelman, and U. Mitra, Eds. Springer Berlin / Heidelberg, 2009, vol. 5516, pp. 131–144.

[194] M. M. H. Khan, J. Heo, S. Li, and T. Abdelzaher, "Understanding vicious cycles in server clusters," in *31st International Conference on Distributed Computing Systems (ICDCS)*, June 2011, pp. 645–654.

[195] T. T. Johnson, J. Green, S. Mitra, R. Dudley, and R. S. Erwin, "Satellite rendezvous and conjunction avoidance: Case studies in verification of nonlinear hybrid systems," in *Proceedings of the 18th International Conference on Formal Methods (FM 2012)*, D. Giannakopoulou and D. Méry, Eds. Paris, France: Springer Berlin Heidelberg, Aug. 2012, vol. 7436, pp. 252–266.

[196] P. S. Duggirala, T. T. Johnson, A. Zimmerman, and S. Mitra, "Static and dynamic analysis of timed distributed traces," in *Proceedings of the 33rd IEEE Real-Time Systems Symposium (RTSS 2012)*, San Juan, Puerto Rico, Dec. 2012.